

AMOS

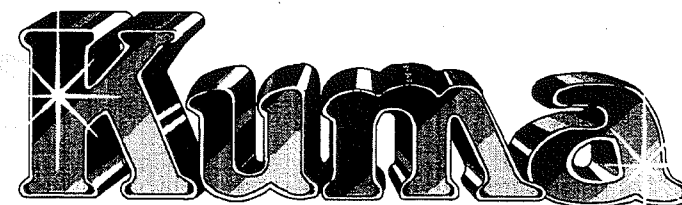
In Education

Anne & Len Tucker

AMOS In Education

By Anne & Len Tucker

ISBN 07457 0225 2

The logo for Kuma, featuring the word in a bold, stylized, 3D block font with a metallic texture and a black outline. The letters are slightly slanted and have a sense of depth.

AMOS In Education

©1993 Anne & Len Tucker

This book and the programs within are supplied in the belief that the contents are correct and that they operate as specified, but the authors and Kuma Computers Ltd shall not be liable in any circumstances whatsoever for any direct or indirect loss or damage to property incurred or suffered by the customer or any other person as a result of any fault or defect in the information contained herein.

ALL RIGHTS RESERVED

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, scanning, recording or otherwise without the prior written permission of the author and the publisher.

Published by:

Kuma Books Ltd
12 Horseshoe Park
Pangbourne
Berks
RG8 7JW

Tel 0734 844335
Fax 0734 844339

Table of Contents

Chapter 1:	Introduction.....	1
Chapter 2:	Where should we start?.....	7
Chapter 3:	Games Specifications.....	15
Chapter 4:	What's on the menu?.....	23
Chapter 5:	Game 1- Eskimo Ice creams.....	45
Chapter 6:	Game 2- Jungle Crossword.....	55
Chapter 7:	Game 3 - Duckshoot.....	69
Chapter 8:	Game 4 - Postman Counting Game.....	87
Chapter 9:	Game 5 - The Amazing Word Machine.....	93
Chapter 10:	A Trip Down Memory Lane.....	99
Chapter 11:	Polish up your Presentation.....	105
Chapter 12:	Graphics.....	115
Chapter 13:	Where to get some help.....	123
Chapter 14:	Where to Send your Programs.....	129
Chapter 15:	Useful Procedures and Other Routines.....	137
Chapter 16:	Brief Guide to Some Amos expressions.....	167
Chapter 17:	A word about the extensions available for amos.....	177
Chapter 18:	Information about your free disk.....	183
Appendix :	Table of Key State Values.....	187

Chapter 1

Introduction

Over the past couple of years, we have been working exclusively on educational software with well known software houses and now we would like to pass on the things we have learnt to other programmers.

We hope that this book will make you think as well as learn something about what goes into the making of an educational program. It is not a book that will tell you how to do everything in steps from A to Z, leaving nothing to your imagination, but one where you are encouraged to follow broader guidelines and add pieces of coding for yourselves to make the programs complete. The graphics are not commercial quality, neither of us is an

Educational
programming is
as easy as A B C !
Or is it?

artist! They are, however, hopefully of a high enough standard to be suitable for Licenseware or to show to a software house as 'scratch' graphics to get your program idea over - after all, they say that a picture can say a thousand words!



It seems to be a common myth that educational software is far easier to write than your average arcade title, but after talking to programmers who have taken on an educational project for a 'break' between games, we have learnt that they have drastically changed their views!

A good educational package contains very many hidden pitfalls which are not instantly recognisable. We hope that this book will guide you

through the planning and writing of programs which will entertain as well as educate the person using it. This type of program has caused the emergence of the word 'edutainment' which we think is rather hard to say and so we shall call them learning games.

Computers are used extensively in schools and children from the age of 3 are allowed to use them as part of the National Curriculum. There are never enough computers to go around so any extra experience gained at home can be invaluable.

**I'm Fed up of
bashing beasties!
Can you play
learning games on
my computer?**

customise the program for a specific need. It is well worth a visit to a local school to see if the teachers would be willing to co-operate with you in the design and testing of a project, after all, they are at the front end and can see exactly what is missing from the software they have.

Most schools have Acorn or BBC computers, but many children will have access to an Amiga at home. We have talked to children while testing one of our commercial releases and were surprised at the number who wanted to do something apart from bashing up aliens with their computers. Many of these children had games consoles, but if your children, brothers, sisters etc. have an Amiga or at least the use of one, you might well be surprised at their response to playing a learning game which is fun as well as educational.

If you are writing educational software for your own use or for that of your family, you can



Amos provides an excellent medium for creating your programs. It has been used by most of the educational software houses in Britain in the creation of educational software over the past three years as well as giving many people a way of helping their family and friends to enjoy learning.

It should be emphasised, though, that this book will be useful to a programmer using other basics, or even another language, as the principles involved in the planning and some of the coding will be common to all languages. All basic languages have common features, it's just the 'extras' which make some basics easier or better than others!

All you will need to use this book is your Amiga and Amos! No extensions have been used in the programs, so there will be no added cost to you.

Now that there are three versions of Amos, it should be possible to use the one which suits you best. As we use Amos 1.36, this book is written with that version in mind. However, as so many programmers now use Easy Amos or Amos Professional, sections will be included, where relevant, to account for the differences between the members of the Amos family.

Amos 1.36, or as it is becoming known amongst programmers, Amos Classic, is at present, the most versatile of the three. This is mainly because there are so many extensions available which will make programming a lot easier. These include AMOS TOME, CTEXT and SPRITEX from Shadow Software, D-SAM from AZ Software and The AMOS COMPILER from Europress.

Amos Professional is not compatible with the extensions which have been written for Amos Classic, apart from Amos 3D which you can use if you update your Amos Pro to at least V1.12.

The AMOS Pro Compiler has now been released and has improvements on the Classic version, the manual states that it is a stand alone program and will compile programs written with any version of AMOS. It does

have extra commands which will be of use to the more experienced programmer, although most of them will not be of use in educational software writing.

We are getting news of extensions which are being written. One, called Atomix is a packer program which looks very promising for Amos Pro. The other is an AGA extension for those who want to make the best use of the A1200's new chip set with Amos Classic.

Easy AMOS was released as an easy way into basic programming. It is AMOS which has had a lot of the more difficult commands stripped out and a more friendly interface added. Both Easy Amos and AMOS Pro have a tutor or monitor which allows you to watch a program as it runs to see just where a problem is occurring. The manual which comes with Easy AMOS is very well written and a lot less frightening to those who have never tried basic programming before. The author, Mel Croucher, has explained things in a very easy to understand way. There have been plenty of complaints in the past regarding the way in which the AMOS manual was presented, but if you take things a step at a time and try to understand what the command means before rushing on to the next, then this will not be so much of a problem.



Read all about it!



There are now more books available which will help you along the way to achieving your aims.

Our first book AMOS In Action takes you through the steps of writing a maze game called Marvin the Martian and is available from Kuma.

Phil South, a well known AMOS writer, has published 'Mastering Amiga AMOS' which is available from Bruce Smith Books.

Stephen Hill's book 'The Game Makers' Manual' is published by Sigma, many AMOS users have written to us commenting that they found parts of this book very useful.

All these approach Amos programming in a different way, and it is really up to the individual to choose the books he or she feels will help the most.

This book, while being about the production of educational software, will include methods of programming which will be of equal use in other areas. The mechanics of writing an educational platform game are no different than writing a platform game which is pure entertainment.

In order to give you the broadest view on programming, we plan to take you on a journey. The disk which you can send for with coupon at the back of the book, contains all the programs which you will read about as you go through this book and we advise that you send off as soon as possible as seeing the finished product will help you to understand what is being talked about.

We have put together a series of programs covering a wide age range and subject matter. This is to cover as many areas as possible and is something which would not be done in a 'proper' package so please bear this in mind when planning out a project of your own.



We start off by giving you the full game designs, graphics and code for three games as well as the menu or linking program. Then we give you the graphics and designs for a forth section so that you can experiment for yourselves. Finally, we just give you the designs for a section and leave you to use your own graphics, or those supplied for other sections, to create your own version of a game if you so wish.

We hope that you will enjoy learning more about programming in AMOS while reading this book. The old saying 'There's more than one way to skin a cat' is very true when programming. Please bear this in mind while you are reading about the way in which we

tackled writing the sections of this book. The methods we have used may not necessarily be the best or the only way that the games could have been written. We have tried to take beginners into account and so have tried to keep the programs as simple as possible.

We hope that the ideas included will inspire the educational writers of the future to produce all kinds of interesting ideas for PD, Shareware, Licenseware and even commercial releases!



Chapter 2

Where Should We Start?

The first thing to do is to decide which age group you would like to write for. It may be the case that you have been asked to write something for a specific child or class at a school, or you are designing a package for a software house, in which case this decision will not be yours.

Most people have an idea of what they want to write by this stage, the hardest thing is getting it organised and in a condition where the game plans can be set down on paper so that if you need help from other programmers or artists, they will be able to read your designs and come up with work which is as near to your own ideas as possible. Just because a program's designs are down in black and white, it does not mean that you cannot change things later on if you find that they are not working out. This, in fact, is what has to be taken as the usual case if you are working on a commercial project - the finished product is very rarely like the original designs! This can be quite upsetting if it is your first project, you perfect the game to the original design, then when you send off your work to the software house which is releasing it, you are told of countless changes which they see are necessary to make it more marketable or a more polished product.

So, put together your designs - more often called the specifications or specs - to the best of your ability, but be prepared for changes. As you code and test a program you will find things that do not work in the program which seemed like brilliant ideas when the specs were written.

The idea of this book is to give you help in writing educational games of several types, this has been done by sticking together several themes as if they were one. This, of course, would not make a very successful product for release as it is going to end up very disjointed. Please bear this in mind when you are planning your educational project the best way we have found is to think up a central theme and work the game sections

around that. After you have decided on the age group for your program, then think of a suitable theme - toy land is great for very young children, whereas a hi-tech space theme might prove more frightening than educational!

Now you have a theme, write down a list of locations or items related to that theme where the different sections of the package can be set.

Also, you need to decide on how the different sections of the package will be linked together, this will be dealt with in more detail in the next chapter.

When you have a good idea of the things you would like to teach, try imagining how they could be taught via the computer using the theme you have chosen. To make this a bit clearer, we'll take the first game in this book.

We have decided that we want to teach shape recognition. The theme is the snowy Arctic complete with Eskimos, penguins and polar bears. For this type of program, the fact that polar bears and penguins actually live at the opposite ends of the earth is not too important as this is a fantasy setting to teach shape recognition rather than animal facts.

Now we have to think up a way of joining the scene with the shape sorting to make an enjoyable game. Write down as many ideas as you can, then decide which one will work the best using the following guidelines:-

1. Will it appeal to the age group it is aimed at?
2. Can you include a level structure to cater for children of differing abilities?
3. Will the game 'flow'? i.e. can you follow the game through in your head, do the actions follow in a logical order?

4. Can the graphics be done using a reasonable amount of memory? If too many graphics are needed, the program will not run on machines with a small amount of memory. These days, one meg can be considered as a minimum, but not everyone will have 1 meg of chip memory, very many will have 1/2 chip and 1/2 fast ram. For a commercial, Licenseware or PD release, this has to be taken into account when you are writing your program. If you are just writing a program for personal use, then you need only consider your own computer's limits.

5. CAN YOU WRITE THE PROGRAM?

This book will hopefully put you on the right road, but if the idea you have is far beyond your abilities, it is no crime to take a step backwards and try something simpler. Always have a notebook handy in which you can jot down your ideas as they occur, we have a book like this, we started writing ideas in it as soon as we had AMOS. It has plenty of ideas in it, more will be added over the next few years, but no doubt most of them will never be seen by anyone else, as game specs or as finished products, but they are there to give us inspiration when needed. The thing is that newer and better ideas always seem to come along before we get to look in the book! If you are a beginner, a book like this will encourage you to learn more as you will be able to look at your ideas every now and then and sooner or later you will be able to say "I can do that one now!"

And Finally....

There are a couple of other things which you will need to think about when you are planning out your program, these are particularly relevant to a program such as this one where the package is made up of several sections, but are useful in any program.

The Great Escape!

In an arcade game, most of the time anyhow, you start at the start and play to the end, the only option you have is which level to start at. In

**EXIT
use in
EMERGENCIES
only!**

other words, you cannot choose to play another game, simply because there is no other game on the disk to play!

In this sort of product, we are offering several choices to the player, so it is only fair that he or

she should be allowed to move around the different games at will.

In your game sections you will need to include an 'escape' function which will return the player to the menu system where he/she can choose another game. The best idea is to adapt the same system in each section so that the user will know the method of quitting a section without constant reference to the manual.

A favourite method is the <ESC> key as this hints at getting out of something! It could just as easily be 'Q' or a special button on the screen which has to be clicked on to quit.

If this 'hot-key' is hit on the main menu, then the player quits from the whole package.

You Have Been Warned!

How many times have you played a game, or been using a utility when you have pressed a key by mistake which has performed a potentially disastrous function?

If you are like me, then quite often!

An example of this is if you are busy typing in a long letter or article and haven't saved for ages - then you hit a function key in error and lo and behold you are told that all your work has been erased from memory leaving you with a blank screen!

This is a bit extreme, but it does illustrate the need for what are known as

- Dialogue boxes
 - Alert boxes
 - Warning routines
- or, as we like to say
- Idiot detectors!

Whatever you call them, some trap which warns the user that he or she is about to do something irreversible is vital. If young children are using your program, it can save a lot of tears if a warning or confirmation is given about what will happen if they proceed.

The sort of thing we mean is as follows:-

The child chooses to leave game one, before doing so, the program gives them a second chance to return to the game at the point they left it by asking,

Are you sure?	
YES	NO

Adults often find this rather frustrating as they think that if they hadn't wanted to quit etc., they wouldn't have asked the program to do so in the first place! There are times though that an enthusiastic toddler will sneak up to the keyboard and press <ESC> while you are glued to the screen. In a case like this, a simple alert box could prevent murder!

In the menu section, a message confirming that the user has chosen to load the game clicked on is helpful, especially if the user is still unfamiliar with the screen layout. Here the message could read:-

Shall we play <i>name of game</i>?	
YES	NO

A general-purpose alert box routine is included on the disk for you to see how to create your own.

You will have to decide for yourself where to draw the line between the point at which these warnings stop being helpful and become a liability!

File selectors

None of the programs described in this book requires the user to load a file from the program disk, but there will be many times when this is necessary.

It should be your aim to get away from the standard, much hated Amos file selector! As we are talking mainly about children using your program, you should aim to keep the file selection process as simple as possible in order to avoid confusion and frustration.

There are now several alternatives available as source code in the PD Library, if you feel that you are not up to using these, then there are different methods available to you.

You could provide a system of buttons which hold the names of the files which can be loaded, this could be made to update as new files are added to the disk as in an art package where the child might save out his/her own work to load in at a later date.

For young children it is a good idea to name the files automatically as they are saved out. This can be done as saving pictures as Pic1.iff, Pic2.iff etc. Another idea is to ask the child to type in his/her name when the Save option is selected, then the child's work can be saved as Janel.iff, Jane2.iff etc. so different users will be able to identify their own work at a glance.

Level Changing

In order to give an educational package a wider age range, it is necessary to include a level system. This means that you can give 3 for 1 value in your program which often means that 2 or 3 children in a family can enjoy the same program played at different levels without costing Mum and Dad a fortune in software purchases!

This idea also applies to other types of games where the player could select a skill level - e.g. In a car driving game seen as Learner Driver, Driver and Advanced Motorist. - In a dungeon or role player game this could give the main character different levels of magical skill - the list is endless.

As far as educational packages go, the level structure is usually planned to allow children to progress as they are playing by using the skills they have at the lowest age catered for by the program and building on these skills as the level increases.

An example of this is for example in counting:

Level 1: Child counts a single set of shapes one by one

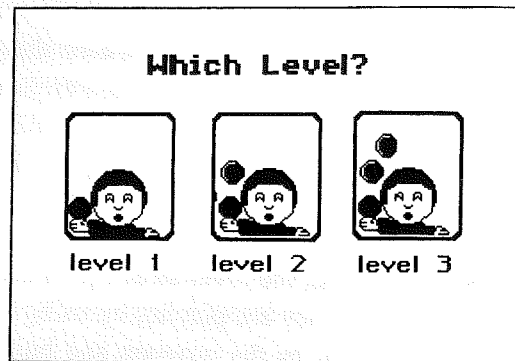
Level 2: Child learns to add two sets of objects together by counting both sets.

Level 3: Child learns to subtract by moving a given number of objects away from the set and counting the remainder.

As with the 'Escape' function, the player should be allowed to move to a different level from within a game by pressing a 'hot key'. We use Function 10 as it is on the opposite side of the keyboard to 'Esc' and is unlikely to be pressed in error.

When this key is pressed, an alert box appears containing 3 choices - Level 1, Level 2 and Level 3. (descriptive rather than imaginative!) As

with a menu, there are many choices of how this is presented. As our program is meant for children, we have made it animate. You could use a text display or have a button which changes level every time it is clicked on. The choice is yours, but will depend on how much memory you have available - an animated inter-level screen may be a luxury you cannot afford!



Chapter 3

Games Specifications.

☞ All programmers need specs! ☞

Even if you have perfect eyesight, you need to organise your programs plans by producing what are properly known as Program Specifications or Specs.

We are giving you the specs for every program in this book so that you can see how to approach spec writing. You will find slight deviations from these specs when you play through the games, this happens every time as you cannot cover *every* eventuality on paper. The specs should be seen as the best guide you can give to a programmer and artist to produce the program you can see in your head.

Even if you are going to do the whole job yourself, it will speed up your programming time if you have a written guide as to what graphics are needed and what bit goes where in the program.

For those who are not coders, but fancy designing games for others to write, it does help to have someone who is a coder to look at your ideas to see if the computer is capable of doing what is being asked of it.

In educational program designing, the finished product very rarely ends up as the original design intended. This is due to several factors:

The Software House will want to put their own ideas in, this is sometimes because they have a particular style for which they are well known and your ideas do not quite fall into the same way of thinking.

The ideas do not work as well on screen as they did in your head so the designs have to change as the coding progresses.

The artist employed by the software house has to put his or her own

interpretation on the ideas, and this means that the graphics will not look exactly as you see them in *your* head!

The artist usually has the foresight to see if the screen layout in the specs will work in the program.

For example, you have decided to put a row of four boxes along the centre of the screen, each of which contains a word. A character controlled by the player comes along and has to select one of the boxes as his answer by jumping onto the box. This might sound fine on paper, but the artist sees problems you cannot and arranges the boxes differently e.g. the words you need to display on the boxes mean that four boxes would not fit onto the width of the screen using the necessary font, so the artist arranges the boxes in two rows of two boxes with the character standing in the middle ready to jump onto the box chosen by the player.

The main thing, if you want to work with software houses on a commercial release is to be flexible in your ideas and not to get too uptight when your brilliant ideas are scrapped in favour of those which may not seem to be as good! This has happened to us on every project so far! The first time came as a shock as we were told that we had more or less total control of the content, but after writing scratch programs (i.e. rough programs written using own temporary graphics) we were told that there were changes to be made which meant that three months hard work was shelved and we had to start again putting in other people's ideas. Now we accept this is normal practice and do not take the changes as personally as we used to.

On the game design side, one project started to be designed 5 months before the go ahead was given for the coding was given. During this time, the game ideas were submitted and scrapped numerous times. Finally, the specs had to be written from vague ideas sent down to us. Now that the project is well under way, the specs are still being changed as some things just do not work!

If you are writing the specs for your own use, you will be able to modify

them as you go along, but if you are programming commercially, you will not be able to change the events of a game without contacting your project manager as the program could well be produced on different machines and so each version will have to be identical.

How to set out your game specs.

The golden rule is to remember that the specs have to be written in such a way that a stranger *could* write the game from them without having to talk to you for hours. An artist *should* be able to draw the graphics from a list supplied on the specs, he or she should be able to visualise the scene from your game descriptions alone.

This is not easy, as we have found out since we have been writing programs, even PD and Licenseware ideas have not always been easy to tie down onto paper.

Step 1:- Package description.

Begin your specs by writing a general overview of the whole package. This is very important for multi part packages such as the one we are emulating here. You should state the age range covered and the subject(s) which are included. Give a brief storyline describing the theme, the menu system and how the various programs are accessed from it. Next give a summary of each section, no more than a few lines here setting out the subject covered and the setting.

Now its down to the real hard work!

Of course, the different sections can be written in any order, (a bit like this book!) then pasted together in the right order at the end. If you get really stuck on a particular section, then put it aside and return to it at a later stage, then if it still does not work, put it in the bin and think up a new idea!

To decide in which order to present your specs, imagine that you are the

user loading and playing the game.

What do you see first? The loading screens - these will include those of the Software House publishing your work along with any copyright notices etc., and a title screen for the game itself.

Next will come the menu system. It is up to you how each part of the package is selected, we'll be dealing with all the possibilities in a later chapter.

The game sections follow next, the order they appear in will depend on the menu system. If you present the user with a list of programs, then it would be logical to put the games in the same order, if you have a map type menu game where there is no strict order listed, then it does not really matter which order the games are presented, although if there are more than one game selected at a certain point, it might be wise to group them together in your specs.

After all the games, you should include other features, e.g. a description of a hard drive installer, if any. There may be a need for the user's progress to be recorded throughout the playing session, how is this to be done and how will it be displayed?

What should be in these specs?

Everything!

Approach the specs, as said before, from a stranger's point of view. Think the program through from the first sight of the screen to the end of the game covering each possible permutation of any event.

Describe the scene you want to set, the approximate size, in pixels of items where relevant. e.g. the main character should be approximately 50 pixels tall or the text panel at the bottom of the screen must be 24 pixels high etc.

Describe in detail how each action takes place in the order it occurs e.g. Penguin enters from the left and moves to the centre of the screen not a penguin comes on the screen - this leaves the artist and coder to guess From which side of the screen does this penguin appear and where does it move to?

Certain events will, of course, be the same for every level and can be covered in a general description. The different levels, if any, should each have their own heading with a full description of the game play.

The Help Feature

The Help feature, if you are including one, should have its own heading, this should be sub-divided into levels if a different help structure is needed for each one. This feature can take several forms. It can be called by clicking on an icon, or after a wrong response. At its highest form, the child could be allowed to get the answer wrong 3 times before being given the correct answer by the program. After each response, added help is given.

e.g. The program asks "Click on two blocks which add up to 7"

The user first clicks on $6 + 3$, The response could be "Try again"

Next he clicks on $4 + 5$, this time the help could be $4 + ? = 7$

If the answer is still incorrect after the third attempt, the program could make the 2 correct blocks flash alternately with a message " $4 + 3 = 7$ "

Rewards

There are two types of 'rewards' - positive and negative.

A negative reward is the response given to a wrong answer. This may be just a text message such as 'Try again' or a graphical response of a chosen object returning to its former place.

One thing must be stressed - in educational programs the negative reward must *never* be funnier than the positive reward as this just encourages the user to aim for a wrong answer.

There are two types of positive reward needed. The first is the response to one correct answer, the second is a reward for getting all or a set number of answers right. e.g. In your program the user needs to get 5 questions right to complete the game. After each correct answer, there might be sound effect of a cheer, and then after the fifth correct answer, he might get a reward of a short animation.

Think of the game as you can see it being played through the eyes of a user of the intended age group and think of things that the user could do, which is not just the things he *should* do.

An example we have come across is that little children are very trigger happy when it comes to clicking the most buttons! This can cause your program to come up with all sorts of strange responses during animated sequences, for instance. You have to try and anticipate this sort of thing by saying that the mouse should be limited to a given part of the screen or frozen at such times.

Graphics lists.

If you are not an artist, it is not always easy to see exactly what will be needed in the way of the number of frames for an animation etc. but you should be able to give the artist a list of the backgrounds needed, a list of the characters along with the actions you want them to perform, a list of bobs/sprites needed as objects for the game and a list of suggestions for spot animations that is little things that happen every now and then, but are incidental to the main program e.g. a tiny mouse running across the screen or the appearance of a tiny spider from a web at the top of the picture. These are things which can be included if memory allows, or cut out if you need the space for vital parts of your program.

Sound effects

Sounds and music are vital to any program. The right effect at the right time can bring a program to life. You may have worked on a program for months with no sounds included, then sound effects are added and the program really takes on a new lease of life! Think of a TV program, maybe an action adventure film, with the sound turned off. It does not have the same effect on the viewer, does it? The right sounds can create an atmosphere which will make the player want to play more, the wrong ones can kill it, so like the graphics, a list should be made of the places where sound effects are wanted and what these should be.

e.g. An engine sound for a car driving and the screech of brakes as it stops. A creak as a door opens, or a bird singing now and then.

Music can be annoying if you are forced to listen to it throughout a game, so there should either be an option to turn it off/on or it should be kept to shorter jingles which are played after certain events such as entering and leaving a game or a reward sequence.

Finally, if you are designing a game to send off to a software house, be sure to state a copyright notice. e.g. © Anne & Len Tucker 1993. on your specs.

If you are approaching a potential publisher of your work for the first time, do not send the finished game designs, but write a summary of your ideas and submit them first along with a short demo of the program if this is possible. Then if the software house is interested, they will make further arrangements to see more of your plans.

Chapter 4

What's on the menu?

Most commercial educational packages are produced as a collection of different games in one package. This is for two main reasons, it adds variety to the package so that the child can move from section to section trying out the different ideas and hence not getting bored as he or she might with a single game product and secondly, from the parent's point of view, it gives better value for money. Full priced Educational programs are not cheap, and so it is seen as giving say, six games for the price of one, even if the individual games are not always as complex as a single game product.

This multi-program idea does mean that you will have to consider some extra points when planning out your package.

1. Are all the games going to be linked around a central theme? If the answer is "Yes" then how are you going to join them together? Do you make a linking game for the player to use to get from game to game, or do you have a static screen with the choices displayed for the user to select?

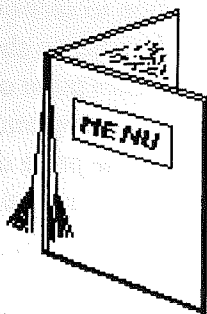
2. If you are going to give a progress chart which records the child's achievements in each game, then you will have to think about how the results of the different games will be recorded and displayed.

There are many ways of presenting a 'menu' from which these games are selected, as we are tackling a multi-part package in this book, then here are some suggestions as to the options we considered before settling on our choice.

The programs in this book have been designed to cover as much variety as possible in order for you, the reader, to get as much from it as possible. It would not have been as useful if the whole package catered for just 4-5

year old children. Bearing this in mind, we do not have a central theme to work with and so have to generalise when it comes to creating a menu system. The principles still apply, even if the scenarios used are somewhat disjointed!

Options

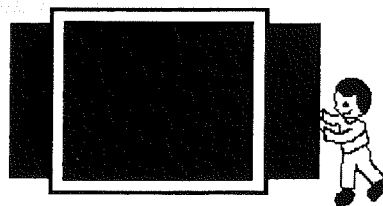


1. A list of the available games set out in text only the user either clicks on the game name or selects by number.

2. A black screen with a box containing the title of each game, or a miniature screen shot. The user clicks on the box to select.

3. A colourful picture incorporating clickable icons which would load a specific game if clicked on.

The name of the game could appear on the mouse pointer as it touches the object.



A scrolling display allows you more room for your menu options

4. A scrolling menu which is two screens wide showing different objects as above. A character moves across the screens until it reaches the item representing a game section.

Clicking the mouse or pressing Fire or Return would select the game.

5. An eightways scrolling game containing various locations representing the game sections. The user would control a character who walks, drives or flies around the map and stops at various places where they can enter

a game.

Take each of these options in turn and see which one suits you best for the package you are producing.

Bear in mind the following:-

- Your coding ability
- The age of the user, it must be attractive to the age group for which it is intended and easy for them to use.

The quality of commercial software is forever increasing and novel ways of presenting a menu will be an advantage.

Here are our reasons for choosing our menu system - the 2 screen wide scroller and for rejecting the others.

1. The text list. This we think is boring and out-dated! It is particularly unsuitable for young children who cannot read. The menu is the first part of the game the user sees after the loading screens and it is an impact point. 'First impressions are lasting impressions' so the song says and it is a point worth remembering!

2. The boxes. This was highly favoured in the past. Both the title only and the mini picture are an improvement on the text list, but are still old-fashioned.

3. The static picture containing clickable items to select games. This is far more acceptable as it can be used to set the scene for the user. We haven't chosen this method as we wanted to give you a bit more than this and the principles of the scrolling screen could be applied to a static one if you ignore the scrolling.

4. We chose the two screen scrolling menu because it looks good and can be applied to the different games in the package. It gives you an idea of how a moving screen can give an added touch to the job of selecting the

games without getting too complicated.

5. To have a linking map game, you really need an overall theme to follow, which this package does not have. It has been used to great effect in commercial products where the main character drives around a map, either playing a simple independent game, or selecting the sections.

Whichever method you choose, you will have to have a menu section in your program. The user gets to the games from this point, but also has to get back to a menu after quitting from a game. It is not practical to reboot your computer between game sections!

With all systems, there should be some system of highlighting the choice made to indicate to the user which game he is about to load.

How this is done is up to you, but some change should be made to the existing display. Games are usually loaded as a result of two actions:

The pointer is over a set area of the screen *and* the mouse button, joystick fire button or the Return key has been clicked or pressed. The following ideas are for indicating that the pointer is over a selected area.

- Change the background colour of the text, or the colour of the text itself.
- Make the frame around a box change colour or flash.
- Make the title of the game section appear on the mouse pointer.
- If using a driving game, make sure that the vehicle has clearly defined stopping points which are only used to load games e.g. car park sign.
- Animate the object to be selected as the pointer passes over it.

There are probably many more ideas, think of as many ways as you can before settling on one method.

It can be useful to have a choice of central characters which the player can use throughout the game. Animal cartoon type characters are neutral whereas if the main character is set as a girl, you will find that boys will be more reluctant to use it, and vice versa. If at all possible, a choice of character should be given to avoid this minor problem.

Menu Specs

We've talked about the type of menu we have chosen, now we need to write the specs for it!

The menu will be set out over a two screen wide, 200 pixel deep display which will at first show the left screen which will scroll over to display the right half as the user moves to the right side of the screen.

The scene is a street in a town. The foreground is taken up with a road while various buildings can be seen towards the top of the screen. These buildings will be used to select the games.

The control of the screen scrolling will be via the mouse or keyboard. As the pointer gets to the left or right edge of the screen, the display will scroll over to show more of the street until the full width is reached, scrolling will then only be allowed in the opposite direction.

The scrolling will be controlled as follows:-



The Mouse

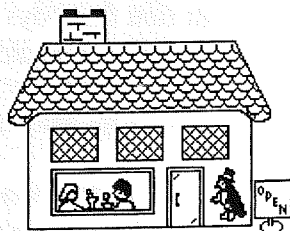
Move the mouse to the left or the right until the pointer reaches the edge of the screen and the screen will scroll over in that direction.



The Keyboard

Use the left and right arrow keys to move the pointer across the screen to make the display scroll over.

Setting the Scene



Scene description

The door of each building will open when clicked on with left mouse button or if <Return> is pressed when the pointer is over the door. Once opened, an alert box will ask the player to confirm that they want to load the game

Building	Game
Ice cream Parlour	Eskimo sorting
Post Office	Inactive, ready for use in Post Office Counting game.
Newsagents shop	Crosswords
Travel Agents	Inactive
Library	Inactive, ready for Word Machine
Toy Shop	Duckshoot

The games will load if:-



Keyboard control

The pointer is moved into position over a door with the right/left cursor keys and <Return> is pressed.



Mouse control.

The pointer may be moved to the door of any building, the game is loaded when the player clicks on the door with the <Left> Mouse button.

Exiting section or program.

To exit any section in this package, the user must press <ESC> on the keyboard. If this key is pressed while the menu is running, then the player will exit from the package completely.

To avoid mistakes, an alert box will be called when <ESC> is pressed, this will contain a message asking the user

"Are you sure you want to leave *name of game*" Yes or No

Yes will return the player to the menu screen and No will return them to the game they were playing.

If on the menu screen, the player will be asked

"Are you sure you want to leave AMOS in Education?" Yes No

<Yes> will return the player to Workbench, <No> will return to the menu screen.

Extra Features.

To make this menu interesting to use, there should be spot animations which will be activated if the player moves the pointer over them and presses Left Mouse Button, Fire button or Return.

Location	Action	Animation
Ice Cream Parlour	Customers sitting in window seat	Eating ice cream
	Penguin with lollipop on shop sign.	Licks lollipop
Post Office	Postbox outside P.O.	Girl standing next to box posts letter.
Newsagent	Chimney	Puff of smoke
	Poster in window	Falls down
Travel Agent	Poster showing train engine	Train moves right to left across the poster, disappears into tunnel at left.
	Plane on shop sign	Flies off poster and disappears off screen. Returns to poster only when shop is off viewing screen.
Library	Clock on front of building.	Hands turn
	Boy outside with a book under his arm .	Opens book and reads it.
Toy Shop	Carousel	Turns round and round.
	Doll	Head nods.
	Upstairs window	Open to reveal vase of flowers.

The Program

The menu program, or as this type of program is sometimes known as a linker, is a very simple program at present. There is a lot more you can add to this to make it a fully working program. This could include the addition of a character who walks across the screen controlled by the player. This character stops in front of the buildings where the mouse or fire button is pressed to select the game. The animation of a boy is included in the bob bank for you to play around with. You could also add more spot animations, such as a bird flying across the screen or create

floating plinth which would tell the player which game is loaded from the building his pointer is over. The alert box system also has room for improvement.

Let us proceed!

Procedure MAIN

The first procedure is called Procedure MAIN and is the central loop of the program. This basically traps all the events whether keyboard or mouse and acts accordingly. It checks to see if the mouse is beyond a certain hardware co-ordinate (NB not screen co-ordinate) and if it is, then the program scrolls the screen. It also checks for the left and right arrow keys with Key State (78) which is the right arrow key and Key State (79) which is the left arrow key. If you can use Key State in preference to Inkey\$, you will find that your program will run faster. It does, however, have slight drawbacks for example if the user keeps a finger on a key without releasing it at once e.g. if the user needs to press F10 to change the level of the game and keeps it pressed too long, you will find that the command echoes. This means that the program will keep going back and repeating itself. This is a problem we had with the commercial programs we have written, but it is easy to cure. All that is needed is a little procedure to stop this occurring. Call your procedure something like Procedure DBSCOD (debounce scancode)

Procedure DBSCOD

```
While Key State (89) :Wend
End Proc
```

This little procedure traps the run of the program until the user removes his finger from the key i.e. the program acknowledges the key has been pressed once but will not let the program continue until the key is released. This procedure can be modified to cover any key by putting a parameter at the start of the procedure as follows.


```

Procedure DBSCOD[NBR]
While Key State(NBR):Wend
End Proc

```

You then can call the procedure putting the number of the Key State you want to trap e.g. DBSCOD[69] will trap the ESCAPE key and DBSCOD[95] will trap the HELP key.

A table of key values is included for you in the back of the book.

There are two conditions at the start of the procedure, the first checks if the '/' key as Key State cannot detect a shifted key on its own. This gives you system information which is useful to put into your program to see how your memory is being used up - chip, fast and total. It is also a good place to print the values of variables you are using which are extremely useful when debugging and checking your programs and to save a tremendous amount of time. You must, however, remember to remove this facility just before you master your finished program as you do not want the information popping up at awkward times when your program is being used.

The second condition checks to see if the ESCAPE key has been pressed, if it has, then the program goes to a procedure called BYE which makes sure that the key was not hit by mistake. If all is OK, then the program exits the loop and the screen fades to end the program.

The Procedure MAIN also checks to see if the user's mouse pointer is over a zone and the button is pressed. If it is, it pops to a procedure SPOTANIM[MZ] to perform the individual anim for that area.

The next condition checks to see if the zone number is greater than 18 and less than 19, in other words is between 13 and 18, if it is, after the anim has been performed, the relevant game is loaded.

The next two conditions check on two special animation areas. If you have clicked on various parts of the menu screen, you will have noticed

that most of the animations are static, e.g. the boy outside the library reads a book, but does not walk away. However, we have included two animations which move on the x or y axes, namely the train and the plane on the front of the travel agent's shop. The plane takes off and flies away and the train travels across the poster and disappears into a tunnel. If either of these are activated, they disappear and are not replaced until that part of the screen has gone out of view. This is more logical as you cannot activate something which is not there and it would look silly to paste the anim back onto its original position after it had just disappeared off the opposite side of the screen! So we made the animation inactive until the screen has moved sufficiently to hide the repositioning of the plane or the train. To make sure that it does not reposition these animations at every pass of the loop, we check on the x co-ordinate of the relevant bob - 4 = train and 7 = plane, to find out if it is at x co-ordinate 900. If it is, the animation has occurred and we need to reset it, if it is not there, then it is ignored.

The last command in the loop is a Wait Vbl. On the surface, this looks like a waste of coding serving no useful purpose, but it does have a very practical reason to be there since Commodore have extended their range of machines. The new Amigas run at faster speeds than the older models, so by putting the Wait Vbl in a loop, you will ensure that your programs run at the same speed on different machines and avoid incompatibility problems on Amiga1200's and 4000's. We have seen some Amos games run on an A1200 become totally unplayable due to the speed at which they work on the new machines. This Wait Vbl cures this problem.

Procedure BYE

This is a very small procedure which displays an alert box, then clears the keyboard buffer using a procedure called CLEARALL then exits the procedure with a parameter. If the parameter=1, (true), then the user did want to do what was selected, if the parameter is <>1 (anything other than 1) (false) then the user changed his mind or made a mistake. As mentioned in an earlier chapter, it is always a good idea to warn the user

that they could be about to leave the program, just in case the key was hit by mistake.

Procedure ALERT[N1\$,N2\$]

This procedure creates an alert box on the screen at the correct position on the viewing screen, not forgetting that when positioning this box the screen scrolls, so we must add the variable SCX to the co-ordinates used when normally positioning the box.

First we find out the width of N1\$. N1\$ is the string used to hold the information displayed in the alert box e.g. Are you sure? If the width is an odd number, then we must add a space to the end of the string to make it even.

To find out if a value is odd or even, or if the width of the string is odd or even, is a simple task in AMOS. To find out if it is odd, use $\text{Len}(\text{N1\$}) \text{Mod } 2$. If this = 1, then the number is odd, if it = 0, then it is even, i.e. if there is a remainder, then the number is odd.

MOD

This is a command that we did not use for a long time after we hit AMOS as we were under the illusion that it was complicated and could not see how it worked. Then, all of a sudden the light dawned and became clear just how easy and useful it is and how many ways it can be applied. This is just one example of its usage. Anytime you want to find the remainder of a division calculation, Mod is the thing to use as it is much quicker than other methods.

Compare these 2 listings:

```
A=17
B= (A/2) *2
C=A-B
```

This gives the same result as:

```
=17
:=A Mod 2
```

Both perform the same function, but the Mod version is far quicker.

Once we've done that, then we have to find out the start position of the box which is the leftmost position of to print. This command will probably look frightening to a beginner, especially if it is seen as a whole. If you break it down into its components, then it becomes relatively simple.

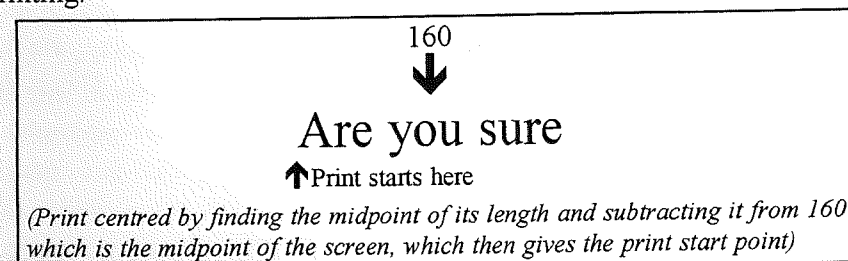
$\text{POS} = 160 - ((\text{Len}(\text{N1\$}) * 8) / 2)$

Let's explain it as the computer sees it:-

At first it works out the length of N1\$ ($\text{Len}(\text{N1\$})$). Now it multiplies that figure by 8. Why 8? The font we are using is 8 pixels wide, so we then get the width of the string in pixels. The result of which is then divided by 2 to find the midpoint, this midpoint figure is then subtracted from 160, which is the midpoint of the visible screen width, to give the point at which to start the printing.

Example:

We want to print ARE YOU SURE (N1\$) Its length is 12. This is multiplied by 8, which is the width in pixels of each character in the font, giving an answer of 96 pixels wide. Divide this by 2 to get the mid point = 48, take this away from 160 = 112. This is the point at which you start printing.



YES	NO
-----	----

The next part finds out the width of the string in characters, we know that the height is 2 characters because we have one line of text and one line containing the buttons. We now open a new screen taking into account the figures we have found:-

```

      * width          height in pixels        resolution
       ↓              ↓                      ↓
Screen Open 3,320,(height+2)*16,16,lowres
       ↑                  ↑
     Screen number    number of colours

```

*Amos does not like a screen opened that is narrower than 320 pixels wide.

The screen has now been opened , but we do not want it to be visible yet, so we hide it with the command

Screen Hide 3

The next job is to reserve some zones ready for the buttons. Note that Amos's zones are exclusive to the current screen, that means that every screen can have its own set of zone numbers which will not affect zones with the same numbers on other screens.

When you open a screen that is not the standard depth i.e. 200 pixels NTSC or 256 pixels for PAL, Amos will automatically display it at top left of the viewing area. Because we want a tidy display, we reposition this new screen centrally to the one it will overlay. This is done in the line

Screen Display 3,,120,,

This will display the new screen in a position that looks central to the

f the screen.

The next set of commands are ones which you should use automatically when you open a screen and should follow the Screen Open instructions.

Fls Off : Flash Of : Cls 0

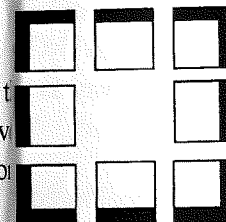
01 The Screen Hide command comes after this as then we can avoid seeing
1 flash of orange screen.

The screen is next cleared for work with

Curs Off : Flash Off : Cls 0 : Get Palette 0

The procedure `_FADEOUT [0]` is called next, we shall talk about this later on.

The time has now come to actually construct the alert box. The listing for this looks rather frightening, but, again, broken down into its separate pieces, is quite easy.

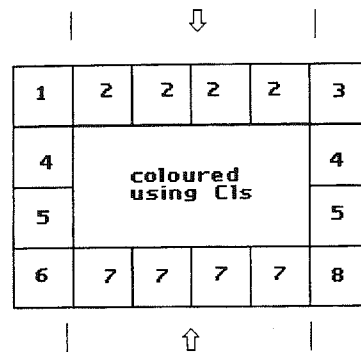


To make an alert box, you will need to draw some bobs as 16*16 tiles which are combined to form a framed box. This method is used because it means that you are not restricted to an alert box of a set size, but have one which will construct itself around any text messages you want, within reason.

The pieces you need are the four corners and four border pieces. An easy way of doing this is to use DPaint and draw a hollow box 50*50pixels in size. You use the inside of this box to draw your framework and colour in the centre space with a colour of your choice, other than the background colour. Now remove the original hollow box to leave a 48*48 square, and cut it into 9 small squares each 16*16 pixels. You can discard the centre one as it is not needed, and you are left with the 8 bits that will be used to construct the alert box in your program.

The box itself is put together as follows:-

Use a For Next loop
to place these tiles



Use a For Next loop
to place these tiles

Top left corner, the top edge is placed using a For Next loop as number of these tiles will vary according to the width of the box, this is completed with the top right corner. The sides are constructed simultaneously from top to bottom, the number again decided by overall depth of the text. The bottom row is added in the same way as top, bottom left corner, edge pieces then the bottom right corner.

This gives you a hollow frame of tiles, the centre is then coloured to match the tiles using the CLS command.

The base is now complete, so we are ready to add the information to it. The text is printed using the procedure called

```
OUTLINETEXT [N1$, 1, 11, 16+OS, 16, 0]
```

This is another example of a parametered procedure. The parameters inside the square brackets provide the procedure with the following information.

Parameter	Information
N1\$	The string to be printed
1	The ink colour for the outline
11	The colour for the inside of the text
16+OS	The x co-ordinate to start printing
16	The Y co-ordinate to start printing
0	A flag which decides if a box is to be drawn around the text to make a button *

If this flag is set to 0, then the program will not draw a box around the text, but if it is set to anything else, it will. The box is drawn in such a way that it has a 3D effect, looking as if it is raised from the surface of the alert box.

We have to find out how many buttons are needed now. This is done by using the '|' character between each word that will appear on the buttons. In this case, we are automatically assuming there are two buttons needed. Try making a third for yourself, it is easy, the text for the buttons is stored in N2\$ and is printed using the OUTLINETEXT procedure, but this time the flag is set so that a frame is drawn around the text to make a button.

Now we do a Show Screen 1 and fade in the screen. The mouse is trapped inside the alert box with the use of the Limit Mouse command. This allows you full control over the user's movements as he is then forced to respond to the message in the box, it also has the advantage of stopping any response to the original screen while the alert box is displayed which can cause other problems.

The program then uses a standard loop to check for a mouse key press which will take the user out of the alert box and react to his decision by exiting the procedure carrying a parameter.

Procedure OUTLINETEXT[N\$,BC,IC,TX,TY,FRAME]

The parameter definitions for this procedure were described earlier as used for the task of printing the alert box text. The following table gives a description of what each parameter holds for use in other programs.

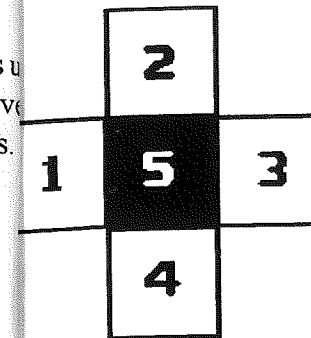
Parameter	Information
N\$	The string to be printed
BC	The border colour(of the font)
IC	The inside colour (of the font)
TX	The start X position
TY	The start Y position
FRAME	Make a button or not

We first of all have to change the writing mode to Gr Writing 0. In this mode, the text will overwrite the information already on the screen without clearing it first, i.e. it will write on top of what already exists.

The width of the string is now calculated in pixels i.e. the amount of characters is multiplied by 8 as each character of the font is 8 pixels wide. The text is now printed using the Text command.

The Text command is used in preference to the Print command because it will allow you to start printing on whatever pixel you want, while the Print command uses multiples of 8 pixels only for positioning.

The outlining of the basic font is achieved by printing the text 4 times using the border colour, in positions 1 pixel to the left, 1 above, 1 to the right and 1 below the position you are placing the text. Next the text is printed again in the main colour in the original position. This gives you an example, white text outlined in black as we use in these programs.



Print your text using the selected border colour (parameter BC) at positions 1,2,3 &4, then print the text at position 5 using the colour in the parameter IC

The program now sees if we require a button to be drawn around the text, if we do, then the co-ordinates of each button is calculated and a box is drawn around each button word. We then exit the procedure.

Procedure LDGAME[NBR]

NBR is the name of the game to be loaded.

Before loading the game, we make sure that this is what the user wants by using the ALERT procedure, but this time with the first parameter we use the word 'Load' plus the name of the game which is stored in the string variable GAMES(NBR) which is set up in the procedure called INIT. If the user wants to go ahead and load the game, we erase a few banks to free some memory, and then run the program name inside the string variable FILENAMES(NBR).

Procedure SPOTANIM[NBR]

This procedure sets up an Amal string with relevant instructions pending the value in [NBR]. The final condition in this procedure sets the Amal Chanel to on if the channel is not already performing.

Procedure INIT

This sets up the displaying of variables ready to start the program. You will notice a Get Bob command in this procedure. In Amos, if you use a Get Bob command after a bob has been activated, it will usually cause the

Amiga to display a Guru message, so if bobs are created on run time, it is better to get as a blank bob here.

This procedure also displays the bobs and initialises the zones as well as filling out the strigs ready for reference in the other procedures.

Procedure INITZONES

This procedure has only one purpose, and that is to set up zones ready for the main program.

Procedure POSITIONBOBS

This procedure displays all the relevant animated bobs.

Procedure _FADEIN[NBR]

This simply fades in the colours at a speed of NBR.

Procedure _FADEOUT

This procedure will either fade out all the colours to black at a speed of NBR or if NBR=0, will turn all the colours to black instantly.

Procedure CLEARALL

This is a standard debounce routine for Inkey\$ and Mouse Key which temporarily halt the program until a mouse button or a standard key is released.

Procedure MVSCREEN[NBR]

This procedure slides the screen to the left or to the right according to user's response. The While Wend loop keeps the program in this procedure as long as the left or right arrow key is pressed and while the mouse is inside specific coordinates on the screen ie towards the left or right edges of the visible viewing area.

This procedure looks extremely jerky when viewed from the Amos editor

when it is compiled it works extremely smoothly.

Procedure INFO

This is a 'disposable' procedure which is put in for use on development only and must be removed before you make your final version of the program as mentioned earlier. It prints out the memory remaining while the program is running and can be used to pinpoint the places where memory seems to be running away without an obvious cause. It is activated by pressing the Shift and ? keys.

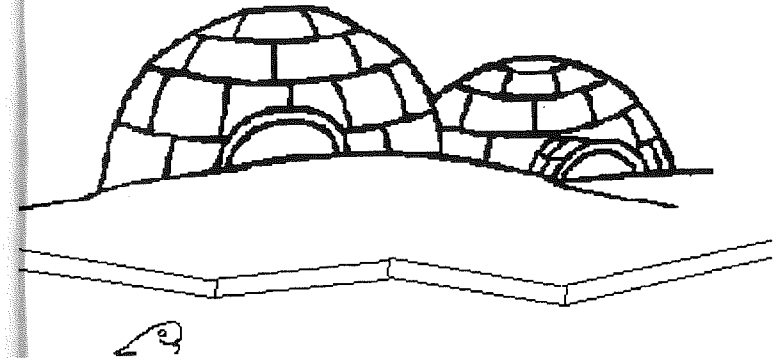
Procedure DELAY[N]

This is a loop that will hang the program until a key or a mouse button has been pressed or until the counter T has reached the parameter N.

In other words if the parameter is set to 100, the program will not go on until either the counter has counted to 100, or until a key or mouse button has been pressed, whichever is the sooner.

Chapter 5

Game 1: ESKIMO ICE CREAMS



Game Specifications.

Sorting game for nursery children

The theme we have chosen for this section is Eskimos and associated items.

The main scene is an ice-cream shop where the customers are Eskimos and the waiters are penguins. The idea is for the child to match up a shape held by a penguin with one from a choice held by the Eskimos.

There will be three difficulty levels in the game:-

In the easiest level, the child will be asked to match a coloured shape from a choice of the same shape of different colours, thus teaching colour matching. He/she will be asked "Find the same colour"

The second level will again show a coloured shape, but this time, the child will be asked to find the same shape from a choice of different shapes in the same colour. Here the emphasis will be on shape recognition.

The hardest level will show a coloured shape for the child to match one from a choice of shapes of different colours, here he/she will have to find the identical shape.

Scene Description

The ice-cream shop is set in an igloo, therefore the walls will be made of ice blocks. There is a counter set in the middle of the screen, it touches the left side of the screen but ends before it reaches the right border, to allow a space for the penguins to enter.

The wall behind the counter is made of ice blocks and has a window looking out through which you can see snowy scenes. A door is situated on the right.

The counter has advertisements pinned to the front as well as a display case which holds some glasses. The floor is also paved with ice blocks.

The customers are Eskimos who are wearing fur lined hooded overalls. They are shown from the chest up only. There are five Eskimos, their costumes should be of different colours to add variety.

Also needed are the coloured shapes and lollipops. Animation frames will be needed for the Eskimos, penguins and lollipops as will be described in the next section.

Game Flow

The game runs as follows.

On initialisation, the scene is deserted. Then the Eskimos pop up behind the counter one at a time. To add realism to the scene, the Eskimos will blink and look around. Each of the Eskimos is holding a shape. The shapes and colours will vary according to the level being played.

A penguin holding a shape enters through the door and moves to the

left side of the screen where he will stand displaying the shape to be found.

The second penguin enters as before, but this time he will waddle from left to right on a random path as if waiting to catch something. The player will now be given a message indicating what they have to do. This will be done to be put very simply or even as a picture as children this young may not read.

The child then clicks on the Eskimo which holds the shape which matches that held by the penguin on the right. The Eskimo then drops the shape he is holding and the penguin catches it. If the answer is correct, the shape that the penguin is holding changes to a lollipop which he then throws back to the Eskimo as a reward. The Eskimo then licks the lollipop. A wrong answer results in the shape being thrown back to the Eskimo it came from ready for a second attempt.

After a correct response, the penguin on the right walks out of the shop and returns holding a new shape. The Eskimos duck down behind the counter and reappear holding a new set of shapes.

Reward Sequence

After five shapes have been correctly matched there will be a reward sequence.

The reward should give the child something new to watch, and should try to make the player smile.

The fifth correct answer will give the same reward as before, then the remaining Eskimos will drop their shapes one at a time to be caught by the waiting penguin. Each of the Eskimos then gets a lollipop. This could be accompanied by animation of the Eskimos' eyes and they could lick the lollipops.

The final reward sequence will be a penguin waiter entering from the left side of the screen carrying a tray with an ice cream sundae balanced on it. This

penguin is showing off by 'skating' on the icy floor. When he gets point halfway across the screen, his feet slip from under him and he on his back. The ice cream flies up, turns over and lands in a messy on his head. This should be accompanied by a jingle and relevant effects.



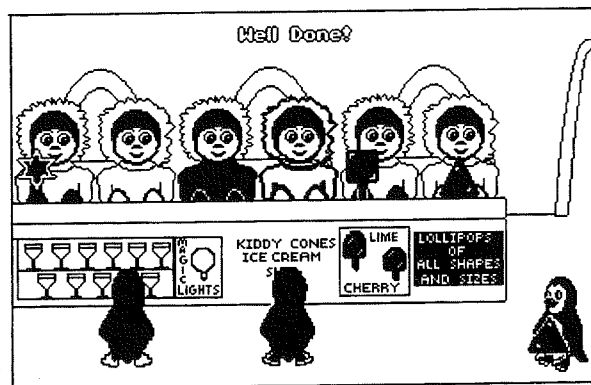
As an added touch, can be random animations. These include:-

- A polar bear or a walrus looking through the window or the door.
- Snow flakes falling outside.
- A mouse or other small creature ice skating across the floor.
- A spider with scarf and woolly hat dropping from the ceiling.

These animations can occur now and then in the reward sequence, the child will not know exactly what to expect as a reward.

The player should be able to exit the sequence at any time by clicking mouse, pressing fire, or by pressing the spacebar.

The Program



of the others in this book.

wait until there is a mouse zone or a mouse click response and then ve the penguin to the location, drop the shape, and check on the shape /or colour and respond accordingly.

only other things in the loop are checks which are made to the QUIT, W LEVEL and HELP keys.

ain there is no help routine in this program, but it should be quite easy apply. The Procedure is already included in the program so all you re to do is put in whatever help you wish to give the user.

Ip examples:

u could use the same responses as the player would get if he got an sower wrong, or you could try to help in other ways such as making the pe on the correct eskimo flash.

Procedure NEWLEV

is is one procedure that with a bit of thought and a bit of work, can be pped out of your program and put into every other section of the book. l you will possibly need to do, is to change the bob numbers if necessary if they appear elsewhere in your program and to alter the image mbers.

The first part of program is the as in any other initialisation pro

We then enter main loop of program which really very si

although the pr appears to be complicated

rst of all we clear the print area on the display screen then we open up new screen just deep enough to hold the change-level panel. We then rn all the colours to black so that the user cannot see all the graphical changes taking place when the panel is being constructed. The panel is constructed in the same way as the alert box we described in the menu section.

nce the panel has been constructed, we print the numbers 1, 2 and 3 to late to each level and set up the zones. Next the bobs are attached. We ould have used the same bob numbers from the displayscreen, but in

AMOS, if, for examples sake, you display Bob 1 on Screen 0, then up another screen, Screen 1, for example, and then try to display B on Screen 1, AMOS can get confused. It seems to assign a screen number to a bob the first time it is displayed. Some of the results we have are corrupted data in the bob image and the total non-appearance of bob i.e. the bob is not displayed. Your X bob, Y bob and I bob will be correct, but it is just will not display an image - it doesn't all occur, but it's better to be safe than sorry.

Now that everything is attached, we fade in the screen and leave display until the user decides which level he wants to play. Once the has been selected, we fade out the change level screen and then slide down to save memory.

The eskimos are now made to slide behind the counter, penguin 1 off the screen followed by penguin 2.. They are all then brought back the new shapes.

Procedure QIT

This procedure is the same as the procedure in other sections allows the user to exit the program.

Procedure ALERT

This is also the same procedure as we use to construct the alert box other programs.

Procedure DROPSHAPE[NBR]

All that this procedure does is to drop the shape to the penguin checking that the limit bob is reset to the size of the whole screen

Procedure OUTLINETEXT

Procedure CLEARALL

The above procedures have been explained in other sections.

Procedure LIFTSHAPE[NBR]

procedure returns the shape to the eskimo either as a shape if wrong a lollipop if right.

Procedure MOVEPENGUIN[NBR]

procedure moves the penguin on the left to catch the shape. We take out the start co-ordinates and the destination co-ordinates and subtract one from the other to find out how far to move the penguin on the x and the y axes.

Procedure CHECKANS[NBR]

procedure simply checks to see if the user's response is the same as the answer. If it is, and the amount of CORRECTS are below 5, then go and get the shape. If CORRECT = 5, then go to the reward sequence.

If the user is wrong, throw the shape back to the eskimo and jump to the procedure called WRONG.

Procedure CRECT[NBR]

procedure is one basically resets the zone that the user clicked on so that they cannot click on it again. It also lifts the shape back up to the eskimo and sends the eskimo back to the place he came from and creates a new question.

Procedure NEWQUESTION

procedure gets the next value in the array according to the value of CORRECT. If CORRECT = 1, then it goes to the first element, if CORRECT = 2 then it goes to the second, and so on. By doing this, nothing is repeated.

Procedure RETURNPENG

procedure sends the penguin back to the bottom left of the screen.

Procedure REWARD

This is where you place your reward sequence. Once the reward occurred, then the program sets up the screen for the next set of questions.

Procedure WRONG[NBR]

This simply gives a different response giving extra help after each answer for up to 3 attempts, then it loops back and starts again.

On the first pass of wrong, we tell the user that the shape he has selected is not the shape that we are looking for.

On the second pass, we tell them what shape or colour they have chosen.

On the third pass, we show them the shape that we are looking for and also make it flash.

The flash is created very simply with a For Next Loop which displays the bob which holds the shape at x co-ordinate 900, which is off the screen and then back where it should be, thus giving the illusion that it is flashing.

Procedure REFRESH

This routine sets the whole process from scratch of creating questions, gathering shapes and then displaying the eskimos and the penguins.

Procedure STZNES

This reserves five zones to cover the five eskimos.

Procedure GOODBYEPENG1 and GOODBYEPENG2

These make the penguins walk off the screen.

Procedure COMEONPENG1 and COMEONPENG2

This is what this does? Of course, they make the penguins walk onto the screen.

Procedure PIKSHAPE

This creates the arrays that hold the answers in the order that they are presented. Then it changes the colour of the bob.

Procedure MAKECOLOURS2

This routine changes the colours of the bob for level 2. This procedure contains 4 very useful little routines. The start of the lines are:-

Deek = Deek

Deek = Deek

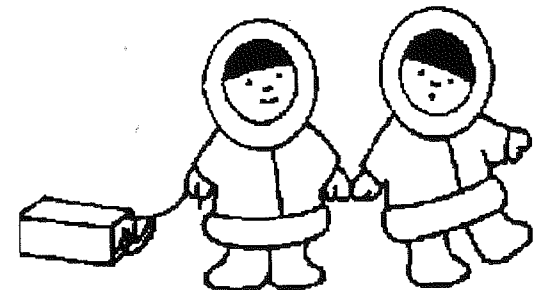
Deek = Deek

Deek = Deek

Use four, in order, find out the width of a bob, the height of a bob, the X hot spot and the Y hot spot.

Procedure MAKECOLOURS

This is very similar to the previous procedure, except it works with a different order of bobs.



Procedure PIKSHAPESL2

This procedure shuffles an array in readiness for the creation of questions.

Procedure PIKSHAPEL3

This is the same as the above, but for level 3.

Procedure PRINTLETTER

Procedure DELAY

These two procedures are the same as those described in other sections of the book.

Procedure INIT

This loads in the data that's required and gets and places the bobs that are needed.

Procedure SHWESKIMOS[NBR,UD]

This procedure performs 2 functions. As a result of the value of variable UD, an eskimo will either come onto the screen or go off the screen. If UD = 1, then an eskimo will come onto the screen, if UD < 1, (anything other than 1), then he goes off the screen.

The effect of an eskimo sliding behind the counter is very effective. It is very simple to create. You simply need to limit the bob to just behind the top of the counter and then move the bob to a position below the counter.

Procedure FOUT

Procedure FIN

Procedure RDOM

Procedure PRINTLINE

These have all been described in other sections of this book.

Chapter 6

Section Two - Jungle Crosswords



crossword game for 7-10 year olds

The program teaches children to find words from given clues, or definitions and place them in a simple crossword grid.

The game will be set against a colourful background of a jungle. The crossword grid will be overlaid onto the backdrop in the form of down and across boxes - one box for each letter of the word.

The puzzle will be generated by the program from data lists stored on the computer.

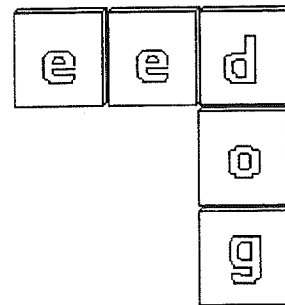
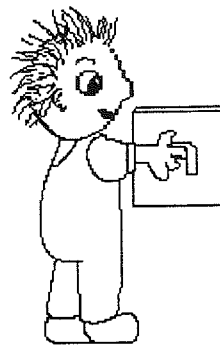
These data lists will have been carefully chosen to make sure that they are suitable for use by the age it is aimed at.

Sample data:

Definition	Word
to leap up into the air	jump
a light fall of rain	shower
soft, red fruit	strawberry
water which is surrounded by land	lake
land which is surrounded by water	island
plant which has spines not leaves	cactus
tall, woody plant	tree
what a window is made of this	glass
my father's brother	uncle
what a camera takes photographs	camera

The program will need to calculate a crossword from these data lists without repeating words and in a form stated in the different level structure descriptions.

The clues will be displayed as the mouse passes over the square which will hold the first letter of the word. This will make the display



cluttered and confusing. Only one clue will be displayed at a time. There will be a bar at the bottom of the screen where clues and any information is given.

The answers will be entered using the keyboard. To enter a word the user will first

all have to click on the first square of the word. The square will highlight and the user will then be able to type in the word which will appear in the message bar. When it is complete, the child will have to press <Ready> and a check will be made to see if the input matches the correct answer. If it is right, then the word is printed in the crossword grid.

If it is wrong, then a series of responses will be displayed as explained below.

First incorrect response

The message bar will display the message "Wrong, please try again".

Second incorrect response

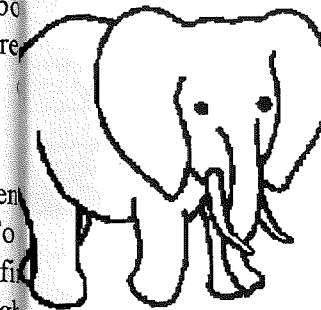
The message will read, "Wrong, this word begins with 'first letter' answer".

Third incorrect response

The message will read "The word is 'answer'".

Help feature.

There will be a help feature available which will put letters on the grid for the word selected. Help will only be available 3 times for each word. As the smallest word will have 4 letters, this will still have one letter to enter. These letters will be selected randomly along the grid.



Level Structure

Level 1

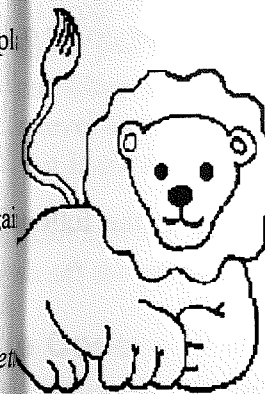
In this level, the grid will be displayed with the initial letters of the answers included and the answers displayed on 'logs' at the bottom of the screen.

The clue is displayed on the message bar as the child moves over the first letter of the word. To answer this clue, the child first clicks on the first letter square to activate it, then on the log holding the answer. The word is then printed on the grid and the log is left blank.

Level 2

Here the initial letters of the words will be displayed as in Level 1 but the answers will

be visible. The child activates a word by clicking left mouse on the initial letter. In this level the child has to type in the whole word as their answer, on pressing <Return> the word is entered in the grid at the selected point.



Level 3.

In the highest level, the grid is displayed empty. The child has to activate the word from the clue only, activating the word as in the previous levels.

	Minimum number of letters in words	Maximum number of letters in words	
Level 1	4	5	4
Level 2	4	6	5
Level 3	4	8	6

Word Banks

The contents of the word banks will reflect the age and abilities of intended users.

The programs in this book are examples of types of educational games and therefore not necessarily full commercial versions. With this in mind, the data sets for this section are smaller than would be expected in a package.

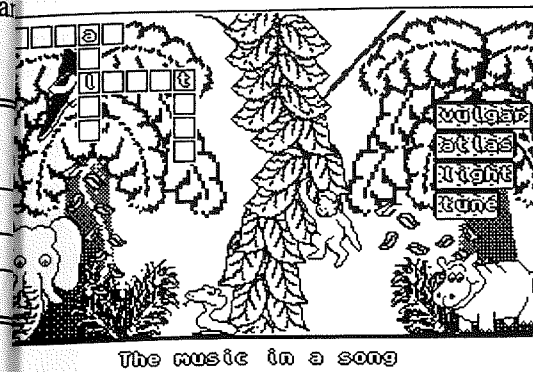
Data Structure.

The data will be entered as ascii (in a text editor) in the following manner.

WORD|CLUE

The '|' acting as a marker between the word and its clue.

program will need to be able to construct a simple crossword from



this data without repeating any words in a single puzzle. The grids should be formed with words placed alternatively across and down and should only intersect at one point along the length of a word.

Program

Our Challenge

In this program, the HELP feature has been deliberately left out for you to figure out for yourselves. Also the inter-level alert box where the player can choose which level to play is not active. The challenge here is to find the inter-level alert box routine in the Eskimo Ice creams game and see if you can cut it out and make it work in this program.

You can also add spot animations and a reward sequence, if you wish, to bring to life the program.

e Code

As usual, we first of all set up the variables and jump to the initialisation procedure. At the start of the listing you will notice the command

```
Buffer 20
```

This increases the size of the variable buffer inside Amos which is needed because of the amount of words we are putting into strings.

Main Loop

This checks for any input from the user and also checks to see if the mouse is over any specific zones whether the mouse button is pressed or not.

The without mouse click check is to find out if the user is at the start of a word or if Level 1 is being played, over a log. If the mouse is over one of these, then it is highlighted by drawing a flashing box around it. It then checks to see if the user has finished a crossword using two variables, CRECT and NXWRD. If CRECT = MAXWORD, then the crossword is finished.

Procedure REWARD

At the moment, this is quite a small procedure. It has been included as a guide to where you should place your own reward sequence, and to the place where you should put what is to happen after the reward sequence has finished. E.g. you should put a jump or a call to another procedure between the lines

```
DELAY[50]
`Your bit and
CRECT=0
```

Procedure CHECKANSL1[NBR,NBR2]

This checks the answer that the user has chosen from the logs in Level 1 against the line they are on in the crossword grid. i.e. the user has to input an answer, NBR=the first mouse click (on the log), NBR2=the second mouse click (on the puzzle). The first thing we do is to put the strings from both mouse clicks into two string variables and we remember the length of the string which we multiply by 13 to get the length in pixels. The number 13 is not chosen at random, it is the distance between the plinths the letters are placed on in the puzzle grid.

Now we have to check to see if the word goes across or down.

using MOD. Because the words were initially placed in order starting across, then down, we are able to deduce that every odd numbered response on the crossword will be across, and every even numbered will be down. Going on the result of this we grab a block of the screen that totally captures the length of the selected word. This is done and then the display back as it was if the user gets the answer wrong. We then have to clear the first letter from the grid as it is displayed for Level 1 and then print each letter of the word they have selected in turn. If the answer is correct, we reset both zones to stop the player clicking on it again, it also prevents it from highlighting. To get rid of the log containing the word, we copy an identical piece of Screen 1 to Screen 0, print the usual 'Well done!' message before incrementing the variable CRECT by 1.

If the user puts in a wrong answer, we print 'Wrong' and we paste down the block which we grabbed at the start of the procedure which returns the crossword to its original state.

The next line is

```
While Mouse Key <>0: Wend
```

This will trap the flow of the program until the user takes his or her finger off the mouse button. This might seem unnecessary, but it will prevent the program from responding as if the button had been pressed several times.

Procedure LEVELPRAMS

This procedure as it stands, is a very simple one, and at the current state of the program is not really necessary. When the player wants to play the game at another level, it means that the variables inside your program will have to be reset to take into account the conditions which have changed in that particular level. This could get complicated, but if all the variables are reset from inside a single procedure, then all you will have to do is call that procedure. At present, this procedure defines how many words are used in a crossword. i.e. 4 words for Level 1, 5 for Level

2 and 6 for Level 3.

Procedure NEWLEVEL

This procedure, as pointed out previously, includes an alert box which will allow the user to select the level they wish to use. This basically replace the line

Add LEVEL 1, 1 To 3

The rest of the procedure simply fades the screen out because we want the user to see any massive graphical changes to the screen. It is very sloppy coding. It then copies all of Screen 1 to Screen 0 and the screen back in. It sets up the level parameters and gets the parameters. You could also add to the level difficulty here by giving different words for each level.

Procedure HELP

As you can see, there is nothing in it! It has been put in and has been called from the main code for you, the programmer to add whatever you would like to include as a help feature.

Some suggestions are, an extra letter could be displayed every time a wrong response is made, this could be printed at the bottom of the screen or in the spaces in the grid. Another idea is to print a letter or two at the beginning of the word on the first wrong, a letter more for a second wrong and the whole word for a third wrong.

Procedure GTINPUT[NBR]

The purpose of this procedure is to get the user's response from the keyboard.

First of all we find out the direction in which the word will be printed as we did earlier, and we find out what the correct word is and put it in a string variable called T\$.

Now we have to calculate where the user happens to be on the screen in screen coordinates. We can do this quite easily using the same co-ordinates that are used for the highlight box routine. These are WDC(NBR,0) for the X co-ordinate and WDC(NBR,1) for the Y co-ordinate. Now we print our prompt line as a prompt and we create a string with every lowercase letter of the alphabet. While we are in the loop that checks for user input we allow them to move around and highlight other words so they can use this routine in one of two ways, either by pressing carriage return or by highlighting or selecting another, or the same, word.

Now we check to see if anything has been stored in K\$, which is the variable where we store any input from the user one character at a time. To do this we use INKEY\$.

INPUT v INKEY\$

You have total control over the user's input, **NEVER** use INPUT, **WAYS** use INKEY\$. It's a lot harder to work with, but is much more controllable.

If you use INPUT, your program will not be allowed to perform any other task until the user has pressed Return to say he has finished.

If you use INKEY\$, then you can allow the user to do other things as well. In this program, that means that the player can highlight a box or select any other word with the mouse key in the middle of typing in the word.

Now we check to see if the letter which has been keyed in is inside the string created with every lowercase letter of the alphabet, and also if the length of the dummy string is less than the number of letters in the word. This ensures that the user will not be allowed to put in a word which has more letters than the number in the target word. If this does not conform to all these conditions, then we add it to the string we are constructing called WD\$, but we do not add it to the end, this would be wrong on the assumption that the user is at the end of the word. Under

normal text engine standards, the user would be able to use right arrow keys, backspace and delete keys to correct the input. The one covered here is backspace, the main reason being that for the age group this is aimed at, the fewer control keys used, the less confusing the program will be. The children would also gain more by having to reposition the word as a whole than by changing odd letters here and there.

We insert the character put in by the user into a string with the line

```
WD$=MID$(WD$,1,PS-1) + K$ +MID$(WD$,PS)
```

What this does.

Take PS, which is the position of the cursor - in an example word

The user has keyed in the word **table** and he now wants to change it to a 'b'. The user would position the cursor on the 'p' which means he will be on the third letter of the word, so PS=3.

The above command would split down into actual transactions, assuming that the second transaction from the user is a letter 'b', as follows

```
Mid$(WD$,1,PS-1) = "ta"
K$="b"
Mid$(WD$,PS) = "ple"
```

The result of this would be "ta" + "b" + "ple" which is inserted.

For overwrite, the line would be

```
Mid$(WD$,1,PS+1)
```

Which would give the result "ta" + "b" + "le"

It is obvious from the above example that the most critical part of the input engine is the position of the cursor relating to the count in the word and also its position in count in pixels. If you are able to keep

control over these two variables, then you are well on the way to success. The engines (i.e. the routines which control any text function) are lengthy and a bit complicated, but are not impossible.

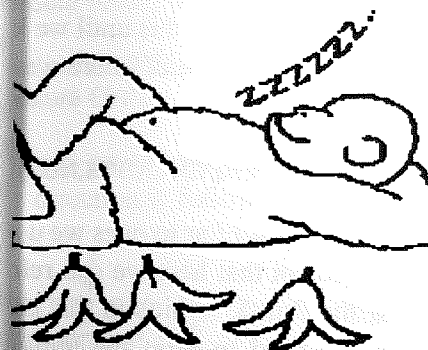
If PS is less than the number of letters in the word, then reposition the cursor to the right.

The text routine checks on CHR\$(13) (return key or mouse button), then the program checks on the backspace key. Because the backspace key is the only control key for editing, we only need to delete the last character of the string and reposition the cursor one to the left.

Inside the while wend loop, we check to see if the user-constructed word (WD\$) is correct. If it is, respond accordingly, if not, also respond accordingly.

Procedure **WRONG[NBR]**

This is the procedure we have staged or stepped the wrong response.



- On the 1st pass of wrong we print the standard message "Wrong, try again"
- On the 2nd pass of wrong, we give the first letter of the word.
- On the 3rd pass of wrong we give the whole word.

Procedure **DELAY[N]**

This is the same as we described in an earlier section,

Procedure REFRESH

This simply copies Screen 1 to Screen 0 and displays a new cross

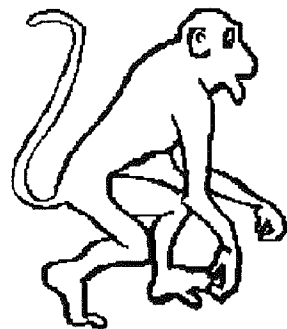
Procedure INIT

This procedure is called only once and initialises the program. Load word bank, unpack the picture which is in Bank 16 to Screen 0 and then hide Screen 1 from view. It also changes all the colour in Screen 0 to black so that we can have a nice fade-in at the start

Procedure LDWORDS

This routine loads in the bank of words and stores them in the reserved Dim'd Strings. One string holds the words while the other holds the definitions. It does this by Blooding the data bank into an Array, then it goes into a loop hunting for an occurrence of CHR\$(10) which is the end of line character.

Between the first and the last occurrence of CHR\$(10) is the next word and definition in the bank and to split the word from the definition we search with INSTR for an occurrence of CHR\$(13) (which is the downline) and we keep doing this until we find the end of the definition. How many words are in the bank - here we have 388.



Procedure _PRINTLINE

This is also similar to other Print\$ procedures. It does not allow repositioning of the string to be printed, but will allow you to char

acter of the text used.

Procedure FOUT[NBR] and Procedure FIN[NBR]

These two procedures have both been described in other sections.

Procedure FIRSTWORD[NMB1,NMB2]

This routine initialises a few variables ready to create a crossword.

Procedure FINDWORD[NBR]

This procedure prints the white box which is the plinth onto which the word is printed and also prints any letters which are visible on initialisation, Level 1.

Procedure _PUTLOGS

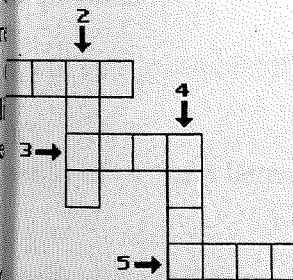
This procedure places the right amount of logs on the right side of the screen well away from the crossword placement. Also it sets up the zones and prints the word which should go on the plinth or log - this procedure proves that the procedure _PRINTLETTER[], although originally created to print an individual letter, will work just as well for a whole string.

Procedure ACROSS

Procedure DWN

These two procedures perform very similar functions. The only difference is the direction it follows. It basically looks at the word chosen for the

crossword and works out a random position for the start of the next word. Once this information has been found, we run through every word in the word bank for an occurrence of the letter we need, if it is found, we add it to the crossword.



Procedure GTWDS

This procedure is the main loop that creates the crossword by alternating calling Procedures ACROSS and DWN.

Procedure RANDOM

This is the same random generator routine which we described in a previous section which gives you a list of numbers with no repeats and no

Chapter 7

Level Three - Duckshoot



Line

The game will be loaded from the Toyshop.

The scene is the shooting gallery of a fairground. The character in this game is holding a gun with which he will shoot at the ducks as they pass in front of him. The player will have a predefined number of shots with which to shoot the ducks and complete a word.

This is a spelling game where the aim is to complete a word by shooting the letters on the ducks as they roll along the conveyor belt.

A clue is given at the top of the screen and dashes under this will show how many letters are needed to make up the word. The letters may be in any order and each time a correct one is hit, it will appear in its correct place.

Level Structure

There will be 3 levels of difficulty which are set by the use of wordbanks.

Level 1

Words of 4 and 5 letters from word bank 1.

Level 2

Words of up to 6 letters from word banks 1 & 2.

Level 3

has words of up to 8 letters from word banks 1, 2 & 3.

Gameplay

The screen opens up to show a duckshoot stall at a funfair, as described in the Graphics list below.



A conveyor belt runs across the centre of the screen. The belt carries a row of yellow ducks. The main character is shown from the waist up and is allowed to move horizontally across the screen, there is no up and down movement.

The character carries a splurge gun with which he shoots at the ducks. A crosshair target is visible between the end of the gun and the ducks. The player uses the crosshair to aim at the ducks. The aim of the game is to shoot the letters of the word which is described in the definition at the top of the screen.

The player is allowed up to 12 shots to complete any one word. The number of shots remaining is shown on the back of the main character's jacket and decrements by one each time the gun is fired.

Game Controls

This game is controlled by the mouse or the keyboard.



Moving the mouse left or right will move the character in that direction. To fire the gun, the player must click the left mouse button.



For keyboard control, the left and right cursor keys are used to move the character. The spacebar is used to fire a shot.

Display

At the very top of the screen, a panel is used to display the definition of the word. The word itself is represented by dashes underneath it. A letter is picked from the relevant word bank and is displayed on this panel. The program then prints an underline character for each letter in the word.

Ducks appear from the right hand side of the screen and move across to the left. Each duck has a letter displayed on its back, every letter that makes up the target word must be there, including repeats of letters which appear more than once in the word. These are mixed up in a selection of random letters and should not appear in the order they are given in the definition. The player is allowed to shoot the letters in any order they wish.

Positive reward

If a correct letter is hit, then the duck gets a splat (like a custard pie) on its head and then sinks down out of view.

Negative reward

If a wrong letter is hit, then the duck sinks without a splat and one of the following messages is displayed:-

'Letter x is not in this word'

'Letter x has been used'

End reward

When the whole word is correctly shot, the flash colours which decorate the sides of the curtains are activated to create a twinkle effect and a message is printed.

'Well Done! Press a key to continue'



When the key is pressed, the ducks that are visible on the screen are to fly away and the screen fades out.

If the player fails to complete the word before he runs out of bullets, the ducks sink out of sight and a message is displayed saying

'You have run out of bullets'

The game loops back to the beginning and a new word is displayed

Graphics

Background screen

This will show the duck shoot stall at a fair. The side edges of the screen are draped with tied back curtains. These curtains have an edge twinkling lights. This is done by using two colours which are nowhere else on the picture and can be programmed to flash in a reward sequence. The two colours are arranged in blocks of alternating colours along the edges of the curtain.

The far backdrop of the screen shows a pond complete with bulrushes. In front of this, there is a conveyor belt. The wheels which move the ducks are visible and animate.

The Ducks:



These 'swim' in from right to left. These are drawn in the style of yellow rubber ducks such as children use in the bath and are plain. The heads animate as they move forward.

For the reward sequence, we need to have the ducks flying off the screen.

Character

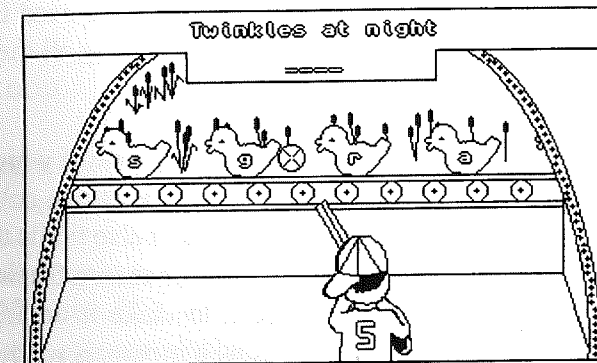
As a small boy dressed in a baseball cap and jacket, he is carrying a specific type of gun which is aimed at the ducks.

A set of numerals is needed which will fit on the back of his jacket. The numerals act as a counter for the number of shots remaining.

The character does not need animation, he just slides horizontally along the lower edge of the screen.

The area where the pond is a blank area made to look like part of the set where the ducks and words are displayed.

Program



There are a couple of features deliberately left out of this program. Why are they? See if you can put them in for yourself?

The inter-level screen is in, but it doesn't react. At present, all the words are used, can you split them up into level groups? There are 97 for Level 1, and 98 in each of the higher two levels.

A general view of the game.



The general idea for this program is to grab a list of words from a bank, choose one of these and split the words into separate letters. Each one of these letters is placed onto a duck and mixed in with letters which carry a random selection of letters. When the player shoots a duck, the program checks to see if the letter is in the word. If not, the duck is destroyed and the player reacts accordingly.

It then checks to see if the user has run out of bullets or completed a word. In this game we have deliberately left out the level structure. The difference between each level should be the complexity of the words. This is reasonably simple to do - it just means that you need to prepare the required set of words and use them, rather than using all the words we do at present.

The first piece of coding is the main initialisation routine which sets the Dims and the Globals, sets the palette and turns all the colours on correctly. It also sets up AMAL and positions all the bobs. It also limits the mouse to a 2*2 pixel area of the screen. There is a reason for limiting the mouse here, even though you cannot see it to make it do anything. It slows down the response if the player is using a mouse as we do not want our main character to fly across the screens if this was an arcade game. This is an educational package which should be steady and slow - although not so slow that the user gets bored.

We move on to the main loop and because we are not using Amos' Autoback and Bob Update systems in this section of the book, it means that we are going to have to do everything manually, i.e. positioning the bobs and changing the screens is now in the total control of the programmer. This can speed up the processing of your

program if it is done properly. The main structure for a totally manual program is as follows:-

1. Will need the following inside the main loop and in every loop that we do the flow of the program.

2. Clear

3. Clear

4. Clear

5. Clear

6. Clear

7. Clear

8. Clear

9. Clear

10. Clear

11. Clear

12. Clear

13. Clear

14. Clear

15. Clear

16. Clear

17. Clear

18. Clear

19. Clear

20. Clear

21. Clear

22. Clear

23. Clear

24. Clear

25. Clear

26. Clear

27. Clear

28. Clear

29. Clear

30. Clear



do this by

1. Resetting all the variables, BULLETS and HIT, for example
2. Selecting the next word in the list

3. Printing the definition of the word in the panel at the top of the screen
4. Displaying a line of dashes to let the player know how many letters are in the word

- Displaying each correct letter as it is shot
- Preparing and positioning the ducks
- Fading in the screen

The second condition checks one string variable against the other. The two variables are WD\$ and TGET\$. WD\$ is the current word being typed to the user and TGET\$ is a string filled with blanks to the same length as WD\$. As the player hits the right letter on a duck, we fill this string with the letter. We see if the user has finished the word - there are many ways of doing this, some are simpler, but this method is used here to teach you a little about string handling. Once TGET\$ = WD\$, we know that the user has finished the word so we go to the relevant reward sequence.

The next condition is to check to see if the player has failed - they will find out if the word is found - just because TGET\$ equals WD\$, does not mean that he has finished. This is done by checking the player X amount of bullets in the gun and displaying this on the jacket. Every time the player shoots, the number of bullets decrements by one, whether or not a correct letter has been hit. Once the number of bullets is less than or equal to zero, (≤ 0), the user has failed and goes to the fail sequence.

We now must find out how the user is interacting with the program - is he using a mouse or the keyboard? It could just as well have been a joystick. Most software houses like to cover all the possible methods of control, even if one or more of them is totally inappropriate for the section in which it is used.

First of all we check to see if the mouse is being used, if it is, we act accordingly, if the keyboard is in use, then act accordingly to that. We also have to check to see if the player has pressed <HELP>, <ENTER> for a new level, <F10>, to do this we use Scancode.

We have very recently found out while writing our last computer project, that Scancode is very unreliable once the program has been running for a while.

led. The problem created is that it causes a condition called a 'key echo' where the key 'echoes' into other sections of the package.

Two entirely separate sections of a package, both of which use the same key to quit the section - let's call them Prog 1 and Prog 2 - We want Prog 2 to go from Prog 1 to Prog 2, and as it should, the alert box to the user and TGET\$ is a string filled with blanks to the same length as WD\$. As the player hits the right letter on a duck, we fill this string with the letter. We see if the user has finished the word - there are many ways of doing this, some are simpler, but this method is used here to teach you a little about string handling. Once TGET\$ = WD\$, we know that the user has finished the word so we go to the relevant reward sequence.

around this is to use Key State (NBR) to replace existing coding. You will have a line like
 COD=89 Then.....
 replace it with
 Key State (89) Then.....

will find that will not echo as it is a one-off test and once pressure is off the key, the value of Key State = 0 no matter how many times it has been put into the buffer.

check is now carried out to see if either the mouse button or the space bar has been pressed which means that the player has shot at a duck.

check to see if a duck has been hit, if so, then we make the duck do what it is supposed to.

next routine keeps on producing ducks as long as the game is not finished i.e. as long as DUCKFIN(0).

next section moves the player 3 pixels to the left or right according to the direction the player has selected.

The next routine is used to replace the definition after a text response, a very good place to put in a nice alert box to make sure that the if the player has hit a right or wrong letter. This is not done immediately really wants to quit the section. as the player will need to see what they have done before it disappears.

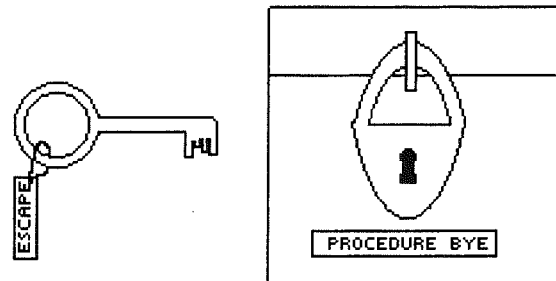
The Procedures

Procedure SCTSCOD[NBR]

This one checks on the main 'special' keys i.e. HELP, ESC and F10 (level) The first routine covers the HELP key - we give the player a word on any press of the HELP key - but the normal way is to have a help feature layered or stepped so that a bit more help is given on every press of the key. Here, you could add to a variable called, for example, 'HELP', so that each time the key is pressed the variable increments by one. If this is done then when HELP=1, then help stage 1 is performed, HELF=2, then stage 2 is carried out etc.

e.g. on each press of the key, one letter is given up to a set amount. you could tell the user that there is an 'a' somewhere in the word.

The second condition is the <ESC> key, which is a simple one. It prints a message



The escape key takes you into Procedure Bye

'OK. Bye Bye! Have a nice day!'

Then it loads the linker program and exits the section.

10 key.

It just increases the level variable by one until it reaches the number of the highest level when it resets to 1 again. By far the easiest method of doing this is to make it automatic by using the command

LEVEL, 1, 1 to 3

It will cycle the variable LEVEL on every press of the F10 key as follows 1-2-3-1-2-3 etc.

Amos users have not got this command, an alternative to this for Amos users would be

LEVEL=LEVEL+1

LEVEL>3 Then LEVEL=1

Then have to tell the user which Level he is playing and put back the position string.

Procedure FAILED

This procedure is called when the player has run out of letters. We tell them what the word was, and make the program hang until the user presses a key when all the letters sink below the conveyor belt. Once this is done, we reset all the variables and set up the display for the next word.



Procedure DNIT

This procedure is called when the player has completed the task. First of all we activate a few flash colours. This is a very pretty effect which we play with two colours which you must reserve before you start

drawing, as they must not be used anywhere else in your picture as they will flash wherever you use them and not just at their in location..

This might be a bit limiting if you are using 16 colours, but there is no real need to restrict yourself to 16 colours for an educational game. The argument used to be that programmers had to cater for 1 meg users, but now that 1 meg is more or less standard, 32 colours can be used more freely. Before too long, the A1200 with its 2 meg memory will be the norm and there will be even less restrictions. You know that your game is going to be exceptionally on bobs, then it will still be worth considering using 16 colours.



In some of the commercial programs we have been involved in, there have been up to 200 images. The Bob Bank which used 32 colours and some of these were 180*90 pixels in size - that's what you can do with memory wise! Just because you are writing an educational program, do not be fooled into thinking there will be next to nothing in it, you will be surprised

to find out that educational games are very graphic hungry, indeed now expected that they will be so.

Once we have set up the flash effect, we let the player know how they are and make the program hang for a while so that they can result. When they press a key, we get a reward sequence, then instead of sinking, we make the ducks fly away.

A short word about reward sequences - There are usually positive and negative rewards, although calling an event a reward when it is because of an incorrect response is a bit misleading, but it sounds better than calling it a punishment section!

The main thing to remember is that the reward for getting things right should be funnier or more impressive than the one after a wrong answer. The idea has to be to reinforce the correct response and encourage

to get more right on the next turn, but we do not want to rub it in. If the child gets things wrong, just encourage them to do better next time.

Remember that a computer is a perfect teacher - it never loses patience and will repeat something as many times as the child wants to. That's true as long as the programmer does not program a nasty surprise if the answer is wrong.

Now reset all the variables and loop again to give another word. What is becoming more common practice in commercial educational programs is that after a specified amount of correct answers, you give the user a graphical reward sequence as a treat. This is great for younger users, there is no harm in giving older users a mini arcade game which can be played if a section is successfully completed. If this sub-game is applied, remember to set a limit to the amount of time allowed as a punishment as you are aiming to improve the user's educational ability rather than their arcade skills.

Procedure HITME

checks to see what letter is on the duck that has been hit. If it is in the word, it adds it to TGET\$ in the correct position. The duck then sinks with a 'splat' on its face and the player is told 'Well done'. If the letter is wrong, then the duck simply sinks down with the appropriate response - if the duck has nothing to do with the word, we tell them, 'No'. If the letter has already been used in the word, the player is told as it will appear to a child that they have hit a letter in the word, when in fact the space has been filled and so is technically wrong. If there is a second occurrence of the same letter in the word, it is treated as if it were the first.



Procedure PRNTINFO[M\$]

This routine clears the print area and then prints whatever in its
This is centred on the screen using our normal formula.

Procedure MVDCK

This part updates all the ducks which are on display, removes
which have gone off the left side of the screen and generates new
to come on at the right.

There is a slight bug in this program which we found when the p
was being tested. Every now and then there is a gap in the row of
We were going to put this right when we decided that the people
reading this book are doing so because they want to *learn* how
and not have everything done perfectly for them. (Are we right?)

One way of learning how to program properly is to try and find
whether in your own programs or in someone else's. So, can you solve
mystery of the missing duck?

Procedure GTDUCK

This is the procedure that constructs each duck. To save time on
it generates 26 ducks - i.e. one with each letter of the alphabet al
it - and stores them for use later in the program. This is done
initialisation and keeps the gap between each turn down to a mini



Notice that the Get Bob command is used in
this procedure. In case you do not remember
what was said earlier, or you skip a
chapter on the menu system, if you
use Get Bob to grab a brand new image
before you display the bobs or you
the risk of your machine randomly locking up. If you are using
on an image that already exists in a bank, make sure that the image
already displayed or you will cause some weird and wonderful effects

Procedure SHOOT

checks for a collision between the target and the duck. It has no
purpose.

Procedure PKWRD

picks the next letter in the array of words and definitions. This
has been deliberately left as boring - i.e. the same word will come
every time the game is first loaded, in this case it will be the word
If we use a procedure such as RANDOM or RANDOM2 and attach
to the order that the words are pulled out, then you will get a
ent word each time the game is played.

Procedure LDWRDS

loads in a bank of words into Bank 9, pulls them out of the bank one
and stores them in an array. If you are using a For Next loop to
the bytes over to the string one by one, then you are going to have
patient as it could take quite a while to perform. The routine we use
is exceptionally fast and can be used in other methods of grabbing
information from data banks. In plain English, it does this inside a Repeat
loop. It searches for the first occurrence of Chr\$(10) which we use
end of line marker. It then works out how many characters are
between either the start of the bank or the last Chr\$(10) and it fills a
any string with that amount of spaces. It then uses the Copy command
py that block of memory to the Varptr of the dummy string which
ally replaces a routine which would copy the information byte by

With the string 'The Cat sat on the mat' you would have to perform
eks to get that string from the bank into the string variable. With this
od, you will only need the one copy command - the difference in
should not need pointing out.

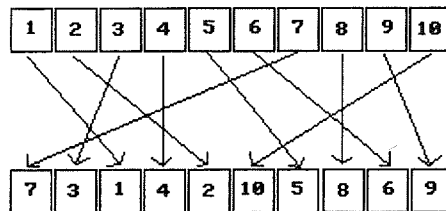
Procedure INIT

This is the basic initialisation routine which is called just the once, then forgotten.

Procedure _PRINT[NBR,NBR2,NB\$]

This procedure has already been described in the linker section book and is more or less the same.

Procedure RANDOM 2LNBR]



Procedure RDOM takes a list of numbers and shuffles it to give a random sequence which has no repeats and no zeros.

routine works is not a true random generator, but it performs its function well enough. It swaps the last number in the array with a random generated number which is the equivalent to a quick shuffle of a deck of cards but only using one card at a time.

Procedure RANDOM[WCOUNT]

This works in the same way as RANDOM2, but with a different set of variables.

Procedure DISBLNK

This displays the underline dashes for the missing letters. For every letter in the target word this routine prints a `_'.

Procedure BTON[X1,Y1,X2,Y2,C1,C2,C3,T\$]

This is a very similar routine to the button procedure in the menu section, used to display an overlay bar with the message 'Press anything to continue'

Procedure WT[NBR]

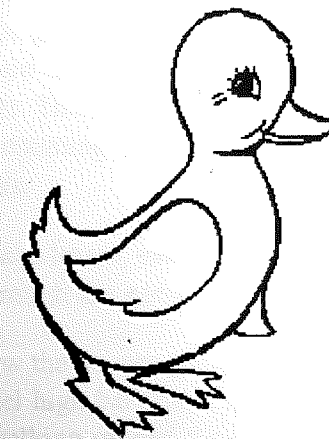
This is a hard wait routine to pause the flow of the program either until the user presses a key or until CU=NBR

Procedure FOUT[NBR]

Procedure FIN[NBR]

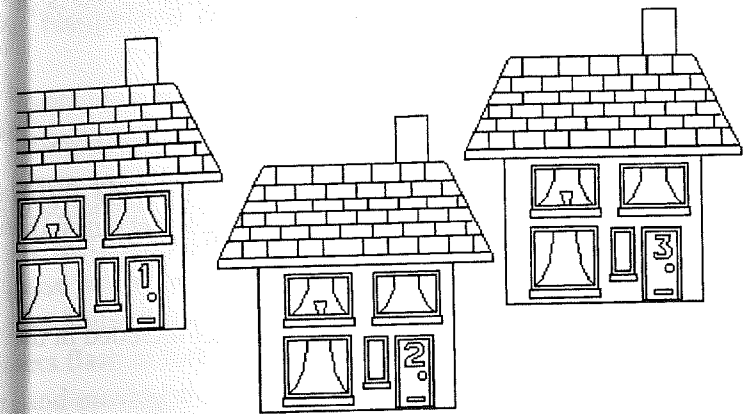
This is one of our routines which was used for one reason only - to be the same as in the linker. It has been used very many times.

In educational programs where you write more and more programs, you will find that there are routines you will need over and over again. In time, you will have a sequence of routines which you can cut out and paste into other programs where there are no major alterations. This will save you a great deal of time while writing programs - routines do not have to be created afresh for every program you write, used routines are often as good as new!



Chapter 8

Postman Counting Game



2 3 L Postman's knocking on my door

A simple counting game for 5-6 year olds.

The game is set out on a two screen wide display which scrolls from left to right. The idea of the game is to show a row of numbered items and a starting point. The child has to count forwards or backwards from this point to find the answer. This will teach the basic principals of addition and subtraction by counting forward and backwards on a visible progression of numbers.

The game will have three levels, each of which will increase in difficulty as the child moves through the game.

The scene should show the houses and pavement. At the very bottom of the screen is a 24 pixel high bar where text messages will be displayed. There is a postman walking up and down the road, he stops in front of one of the houses and looks around. A question will be asked at this

point.

As the houses are so small, the numbers cannot be displayed on the mouse pointer moves over each house, its number displayed on the pointer itself changing as it passes over a new house. The house can be selected as an answer if Left Mouse button is pressed when the number is being displayed.

Level Structure

There will be a very simple level structure with the only difference between each level being the range of numbers used as set out in the table below.

All the houses should be active with the numbers not included in the range being treated as wrong answers.

Level Ranges

Level number	Number of houses active	Answer range	Maximum step
1	10	1 - 10	3
2	15	1 - 15	5
3	20	1 - 20	10

Example:

The Postman walks to house number 5. The message reads

"This is 5, School Road"

The letter is for the house 3 numbers higher"

The child should then click on the house he/she thinks is correct. The Postman walks to the house selected and according to whether the answer is right or wrong, the following occurs.

Positive reward.

If the child gives the correct answer, the Postman walks to the house and up the path to the door which opens to show the person who lives there.

Negative reward.

If the child gives a wrong answer, the Postman knocks at the door, there is no answer so he walks back to the starting point where the question is repeated.

If the child gives a second incorrect response, the postman returns to the starting point where the text bar displays a row of numbers, starting with the number of the house where the postman is standing, in this example 5, and finishing with the house where he should deliver the letter, here number 8. These numbers flash in turn accompanied by a 'ding' sound effect starting from the first number and moving through to the 8 as if counting 3 numbers forward.

If the question had been asked to find the house with 3 numbers less than the starting point, then the opposite would occur.

The numbers 2 3 4 5 would appear and the count would start at 5 and count backwards.

If the child gives a third wrong choice would make the correct house number flash and the Postman walk to it automatically and deliver the letter as for a correct response.

Final Reward

After 5 letters have been correctly delivered, a postvan enters from the left and should be on the correct road level for the number of the house chosen, and stops at a point just before a house chosen at random. A Postman gets out of the van and walks up the path to the house and knocks on the door. The postman leaves a parcel by the door for the person who answers the door, and waves as he moves back to the van and drives off to the left.



1 2 3 4 5 6 7 8 9

Graphics List

A 2 screen wide backdrop of a street containing 20 houses. The houses are arranged in 2 rows, the road loops around like a race track. There are several types of houses both modern and cottage styles, some detached, some semi-detached and a terrace.

Postman walking left/ right carrying a letter in his hand.
Postman front and back view walking up and down path
Several characters who will answer the door.
A parcel for the final reward sequence

Spot Animations.

Face at window
Dog at window
Cat on window sill



Sound Effects



Postman whistling	Knock at door
Door opening	Dog barking
Van engine	Ding for counting in wrong response help

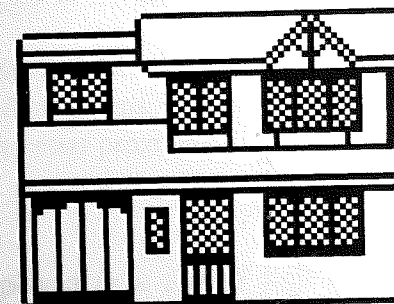
Program



is the time when you are left to fend for yourselves! Well, almost, now.

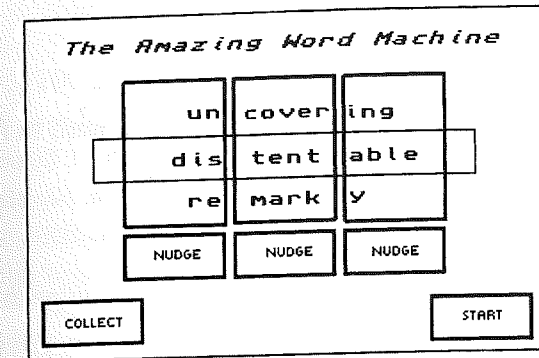
you are given the specs and the graphics for a game, but no code to show it works.

Remember to take each part of the program a step at a time, break it into smaller elements and tackle each one in turn.



Chapter 9

Level Five - The Amazing Word Machine!



is where you are left even further on your own! Here you only get game specs - no graphics and no Amos code!

nope that you will be able to write a version of this program for yourself from the description given. It does leave a lot to your imagination or as graphics, but we have not worked on a project yet where the specifics were exactly as we had imagined they would be!

following game specs are as complete as possible, but should not be as hard and fast designs, include any extras you feel are necessary and make this into a better program!

Overview

The game is displayed as a sort of fruit machine where the user is presented with three reels which 'spin' around at the touch of a button and stop at a specific point. If the reels match up, then the player has won, if they do not, then he has lost his money.

This is a word game where the reels hold different parts of a word, a prefix, the main word and a suffix.

When the player presses 'START', the reels spin and when the only the centre reel contains a word. The other two reels have 'nudged' into place to form new words from a choice of prefixes and suffixes given, the reels which are active are decided by the level played.

Level 1

Here the main word is given and the player is allowed to nudge the left hand reel to add a suffix to make a new word.

Level 2

Here the player is given the main word and has to nudge the left hand reel to add a prefix to form a new word.

Level 3

In this level the player can nudge both the outer reels to form a new word.

Example

REEL 1	REEL 2	REEL 3
UN	STAND	ING
DIS	ACT	ED
UNDER	CHARGE	ABLE
OVER	COVER	SHIP
RE	MEMBER	OR
PRE	ARM	Y

Examples:-

In Level 1, the player is given 'ACT' as the centre reel. He must use the nudge buttons to find a suitable ending - here the suffix 'OR' to make the new word 'ACTOR'.

In Level 2, the main word given is 'STAND' and the player must nudge the prefix 'UNDER' into place to form the new word 'UNDERSTAND'.

In Level 3, the player is given the main word 'COVER' and he is able to nudge the left reel to find the prefix 'DIS' and the right reel to find 'ED' and make the word 'DISCOVERED'.

Layout

The whole 320*200 screen is taken up with the display of a machine which is based on a fruit machine. There should be three windows. Each window will need to be large enough to display a word of up to six letters. They should also be deep enough to display 3 words at a time, the centre position being highlighted by a box to show the user that this is where the word has to be lined up.

prefix	main word	suffix
DIS	COVER	ED

The left reel should be right justified, i.e. lined up to the right hand side of the box/reel. The word in the centre reel should be centred and the word in the right reel should be left justified, i.e. lined up to the left side of the box.

In levels 1 and 2, the inactive reel should be filled with a pattern and should not spin.

There should be a large button on the bottom right of the screen with the word 'START' printed on it, this should be made to indent or change colour when it is clicked on and will start the reels spinning.

There should be a 'NUDGE' button underneath each reel so that the player can click with the left mouse to nudge the reels down and click with the right mouse to nudge the reels up. Every time the nudge buttons are clicked on, the reels will move one word in the relevant direction. When the player is satisfied with the word he has made, then he should click on a 'COLLECT' button which should be situated in the bottom left corner of the screen.

There should be a score panel at the top of the screen which will show the player's score. This should start off at 10 and decrease by one every time the player presses the start button. If the player gets a word wrong then his score is increased by 10 if he gets it right on the first attempt, by 5 if correct on the second attempt at the same word. If he gets it wrong, then his score is reduced by one for the first 2 passes of wrong, then after the third pass of wrong, the reels will spin automatically and show the correct word.

Final Reward



If the player's score reaches the set target score - then there should be a sort of special reward display is drawn on the screen then you could make the machine flash and the player had hit the jackpot.

Data structure

This will have to be very carefully thought out as you will need to match main words with a prefix and/or a suffix.

Data could be stored in the following way:

Divide the data banks into three levels so that it will be easier to work with.

Level 1 data: Here you will need to create lists of words and prefixes with a character dividing one from the other so that you are able to pick out the parts needed to be placed on each reel. In all cases where there is more than one possible correct answer, then commas are used to separate the words.

Level 2 data: This is done in the same way, but with the '|' dividing the words from its suffix.

Level 3 data: This time you will need to choose two dividing characters so that the word must be divided from its prefix and suffix.

Chapter 10

Tip Down Memory Lane

Ways to make your programs more memory efficient.

As you progress from writing small programs to full games or utility programs, the need to save memory will become more apparent. If you have more than 1 meg of memory, then it will be all too easy to get lost away whilst coding and end up with a 1.5 meg monster program - sounds impressive, but the vast majority of users have 1 meg of memory, so very few people will be able to run your program, it is for this reason that there are very few Commercial programs on the market using this sort of memory. The exception to this is applications which chew up tremendous amounts of memory - some landscape generators need 3 meg minimum!

With the advent of the 1 meg standard Amiga, restricting your programs for 1/2 meg users is not so important, and now that the Amiga comes with 2 meg of chip memory as standard, the temptation will be here to cater for these machines rather than the 1 meg A500's and A1000's. It is a fact that there are more machines of 1 meg and above now than 1/2 meg, and these are getting fewer and fewer as the old machines are phased out when people upgrade to the newer machines. The ease of memory upgrades for older machines and the reasonable cost of new Amigas mean that memory restrictions will be lessened for a while, although by the time that 2 meg chip is standard, the average 'serious' user will more than likely have 4 meg or more!

There are times though that a software house will demand that a program will run in 1/2 meg, even if this means tripping out sounds and graphics if the machine being used has only 1/2 meg of memory.

The following tips have been learnt during work on commercial projects over the past 2 years.

1. Whenever possible use 16 colours in Lowres

If you are used to using 64 colours, then this will seem very restrictive! We found the change from 32 to 16 colours difficult at first, but now it's easy, as long as you choose your colours carefully, you can get some great effects. It may surprise you to find that many commercial games are 16 colours, specially 'Cutie' educational packages - Fun School 4 is an example of this. It may speed up the running of your program slightly.

2. Use an NTSC screen. i.e. make your display 200 pixels

This saves memory as the program needs to hold 56 lines less data. It has the added advantage that users in the USA will be able to use your programs. It must be rather frustrating for them to find a program to find that the controls are hidden. Apparently, there is a Fixer program available in America, which works with most things and may make it more difficult to use your program?

3. Set the Sprite Buffer to 18.

This can be used if you are not using sprites or your biggest sprite is deeper than 16 pixels. This does not mean that you cannot use sprites which are not the same thing. You will notice that the AMIBIOS disappears from the editor screen when you do this, this is because the buffer is bigger than the limits set. This tip saves 10k in the run-time memory of your program. It doesn't sound a lot out of your available memory, but it could be the difference between your program running or crashing.

4. Aligning Bobs for animation.

Rather than increase the size of your bob when animating, try and move the hot spot of the bob instead. This is always possible, but when you can do it, it saves lots of memory.

ing Mask.

You have a bob that is a multiple of 16 pixels wide, and is a solid colour (i.e. No gaps) then you will not need to use a mask.

Opening Screens.

When you open a screen, don't close it again. It is more memory efficient to keep it open, clear it and use it again.

Packed Pics.

It is better to pack your screens using the Spack command rather than using the IFF screens.

Squash bobs.

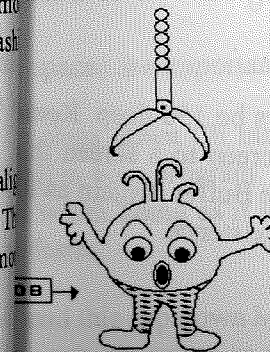
If you have a large amount of bob images which are not all displayed at the same time, then you can Squash them and just pull out the ones you need at any one time.

Sprite Buffer.

You have to use the Set Buffer command, then set it to the amount of memory you need, don't add any for good measure.

10. Less Bobs.

Try to use as few bobs as possible. Don't use lots of Bob numbers just for the sake of saying 'My prog has hundreds of bobs!' Reuse the bob numbers once they become obsolete.



11. Amos Config.

When you run the Config program you'll find a section where you have to set the number of bobs used. This is somewhere around 60. When you have finished your program, calculate the number of bobs used, add 50 to be sure, alter the setting in the Config program and save the program. This is best done just before you compile your program.

12. Screen Size.

Only open screens to the depth needed. For example, if your screen is lowres and 150 deep, and your control panel is medium pixels deep, then you need only open your display screen to 150 and your panel screen to 50 pixels.

Remember though, if you open a screen to for example 150 deep, then you have to load in or unpack a screen which is deeper, the screen will return to a larger size, this is because both the Load IFF and Unpack commands return the screen down and open a new one. Take our example display screen to 320 wide, 150 deep, 16 colours. To make sure that it unpacks to the right size every time, go into Direct Mode and load in your Iff screen. Then load in your display screen to 0 by keying in the following:-

```
Screen Open 1, 320, 150, 16, Lowres : Curs Off
Flash Off : Cls 0
Get Palette (0)
Screen Copy 0, 0, 0, 320, 150 To 1, 0, 0
Spack 1 To 16
```

Now when you unpack Bank 16, it will always be 150 deep. Remember that if you unpack a screen or load an IFF screen to a screen which is Double Buffered, then you will have to Double Buffer it again. Unpacking and unpacking destroy the Double Buffer in existence.

Screen Copy.

You have to use Screen Copy at any time in your game/program then do it. You can copy the whole screen unless you have to - just copy the area you need.

Data

Keep your Data lines to a minimum. If you have hundreds of lines of data, questions etc., then it is much cheaper memory wise to store it as a string and peek out the information as needed.

Workbench and Cli Shutdown.

Another good memory saver on runtime is the shutting down of Workbench and Cli. Doing this involves 2 stages.

1. In your program use the command
`ntcall(-78)`
This turns off Workbench.

2. 1st stage - on your **RUNTIME** disk make sure that the **RUN** command and the **ENDCLI** command are in the C Directory. In your Start-up routine, let's assume your game is called Program, this is what you do

```
> nil: program
cli
```

Spaces are important, so put them in!

2. 2nd stage - Cli shuts down Cli but will open up Workbench, hence you need the command in your program. This need only be done on your runtime disk. It is worth doing as it saves 30-40k and makes your program smoother and faster.

At the end of this trip down Memory Lane, here is a little routine which we put in our programs so that we can see how much memory is left, it can

also show where the memory is being gobbled up!

Basically if you press '?' (i.e. <shift>+?) a screen opens at the top of the display which prints out available memory. Pressing any key gets rid of it. If you use this when you are running your program you might be surprised where the memory is being lost.

Put the following in your main loop. If you are using your own memory variable, then use that, if not, do the following....

```
K$=Inkey$  
If K$="?":MEM : Endif
```

Now put this in with your Procedures.

```
Procedure MEM  
Screen Open 7,320,30,2,Lows  
Print Free + Chipfree + Fast Free  
Clear Key  
Repeat  
K$=Inkey$  
Until K$<>" "  
Screen Close 7  
End Proc
```

Remember to remove it from your finished program to save memory, but also to avoid a strange response if the '?' character is typed in the program by the user.

Chapter 11

Polish Up Your Presentation.

Does program 'A' get accepted for Licenseware or released as commercial while program 'B' is classed as Public Domain, when both are basically the same thing?

I have learnt a lot over the last 3 years whilst writing for Licenseware and also more recently whilst working on commercial packages.

Of all, you do need a good idea. A brand new type of game or program will obviously stand more of a chance than an Amos rewrite of the Invaders.

When you do decide on a topic that has been done before, which is admittedly, if you are writing educational programs, is often the case especially with English and Maths, then you must come up with a novel approach.

A blank screen which merely has a sum printed on it in the default Amos font will not impress anyone, although if it is the first thing you've made the Amiga do for you, then you have a right to be proud.

Even if your maths program was set inside a fun to play game, had colorful graphics and a reward sequence for getting the sums correct, then it is more likely to attract attention. Whatever the subject of your program, the main aim has to be long-lasting appeal for the user. No one wants to pay out hard earned cash for a five minute wonder, so think out what you can put into your program before you plough into the coding.

Although we think that Amos is the best thing to hit computers since the Commodore chip, and we always make sure that Amos gets full credit, there are a few things that it does which will give the game away if you want

to show someone your program before telling them what was
create it. There are still Software Houses out there that will
Amos as a language for their products, although these are getting
Most publishers of educational software are Amos friendly!

The first things to get rid of are the horrible orange screen
cursor ghost in the corner that pops up at the start of a lot of programs
All you have to do to get rid of this is:-

```
Screen Open 0,320,200,16,Lowres  
Curs Off : Flash Off : Cls 0
```

This will give you a 'clean' start to your work. It's good practice
this as it can save problems on RAMOSing or compiling
program to disk. Amos defaults to Screen 0, but if you force the
display yourself, then you are in full control of your program
start. Curs Off: Flash Off: Cls 0 done in that strict order will solve
other problems

We use a screen depth of 200 as there will not be any problems
American users if the program is taken to the USA (or Japan -
matter!) Don't say that your program will never go there so
matter, even PD disks are sold around the world - we know, but
send them out to distributors!

Screen wipes and other special effects will make the menu
screen to screen more professional. If you cannot write these for
there are plenty to choose from in the AMOS PD Library.

Instructions must be clear and easy to read. Put these as a README
which is displayed on loading (with PPmore or similar) or if
disk/memory space as part of your program. Commercial releases
printed manuals, but this is not necessary for PD or Licenseware.

Be sure that the program WILL run in the amount of memory

1/2 meg programs should RUN on a 1/2 meg machine, some
names will load, play for a while then crash out of memory, so test
highly. If in doubt, say that 1 meg is needed or at least highly
recommended.

Amiga Family has now grown to include the A600 and the A1200
are a couple of other things to be taken into account if you are
ing a program using one of these machines.

A500 only has 1/2 meg of Chip memory whereas the A600 has 1
Chip and the A1200 2 meg. This is great, but it means that both
machines have a greater amount of memory available for graphics
found that the A500's owned by thousands of potential customers. So
warned is forearmed!

Remember that the Amiga's memory 'leaks' and a bit is lost here and
during the running of a program which cannot be got back,
before you should thoroughly test your program for a likely memory
out on a machine with a smaller memory capacity than the one the
program was written on especially if you have over the now standard 1
Most Amos programs, games, educational etc. will need 1 meg
ss you have the knowledge on ALL the possible ways of saving
memory.

One thing that really makes a game come to life is the addition of sound
effects and music. Amos does not allow the editing of a Sam bank, so
is easier left till last when you can make a list of samples you will
need and put the banks together in one go. Easy Amos, however has a
verb Sam Bank editor which is very flexible and user friendly.

Our main interest is in educational software, our knowledge on what
makes a good game or a great utility is rather limited to our own
licenseware releases and programs we use ourselves.

We have put together checklists of things to ask yourself after your
program is completed, it is not to be taken as law that if all the points

have been covered, then your program is OK, but as a guideline you on the right path. These points, whilst geared towards educational programs, have many points which are relevant to other programs.

Things to think about.

☞ Are the graphics suitable. For very young children, a cartoon style is the most suitable, futuristic spaceships with blazing would not be suitable for nursery children, but may be a way of making a subject fun for older children. They are the first thing to hit the user's eye, so they have to be good. However flash your coding, pin men graphics will not do. Graphics can however hide the odd inconsistency in quality but it cannot hide everything!

☞ Have you included a level structure? This can take two forms: a definite choice is made to play the game at one of the levels and the program stays set at that level until the user changes it.

or the game is played on an automatic progression from level to another as each is successfully completed.

If the second is chosen, will the player be allowed to start at the stage he has reached, or will he have to start at the beginning? This point is more relevant for an arcade game than an adventure than educational games, but someone might come up with an idea which involves a long 'journey' through the educational program, learning as you go rather than the sectionalised games described in this book. In that case it would be cruel to expect a child to play for 6 hours non-stop to complete the program!

Is it possible to finish the game inside the time allowed or the number of lives or tries allowed? The timing allowed for the completion of any type of program is important. The only way that this can be tested is to play the game from start to finish. If a timer is used, or there are many levels to play, you can cheat by putting in a temporary trainer which allows the skipping of levels or gives infinite time so that you will be able to check if all the conditions in the program work properly. Get friends and family to help you at the testing stage. It is vital that any bugs are noted down, with info saying where and when they occurred and what the player did to create the error. A bug report saying 'It crashed on Level 2' will not help you to debug your program, whereas 'It crashed when I shot the green monster as it passed over the spaceship going right to left' could give you a hint of what went wrong.

☞ Is all your action smooth? Does the joystick/mouse control work properly? Is your collision detection accurate? Does the program react properly when a button is clicked on or when something is 'shot'?

☞ Is the score routine fair, have you left enough room in the score display for someone who will score a mega number of points, if not, are you giving too many points per hit etc.

☞ In a maze game, you must be allowed access to all parts of the maze, it is quite easy to make parts of the maze inaccessible to the player.

☞ Test it on children of the relevant age. Do not leave things to guesswork. You might think that you are playing the game in the same way as a 7 year old, but it is surprisingly easy to under or over estimate the abilities of children!

☞ Don't feel embarrassed at using reference books to check up on facts! Just because you are writing a program to teach spelling,

don't assume that your spelling is perfect, this type of has you doubting your own abilities!

☞ The main thing that you *must* keep in mind here is the you are trying to teach something or reinforce a child's knowledge, therefore **you must be accurate!!!** A child believe that computers are always right, not the truth. A computer can only do what it is told! If it says $2 + 2 = 5$ they might well go on believing it!

English programs are very data hungry. You must make sure the best of your ability that there are no spelling mistakes, tell someone else to check the data with a dictionary if necessary. Mistakes should be rectified before the program is finally released.

Definitions or clues should be clear and if there are two possible correct answers to an anagram for example, then this should be covered. e.g. if the letters TSAR are given then possible answers could be STAR RATS and TARS. It is cruel to say that a correct answer is wrong unless you gave a clue for the one answer you wanted to find, e.g. It twinkles in the sky at night - STAR would then be correct but RATS would be wrong. Yes we've been guilty of errors, grammar etc. in some of our programs, but we have always tried to get them right once they have been discovered.

Data accuracy would be very important also in geography, history or anything else that deals with facts. Remember to put in capitals for proper nouns i.e. place names etc. Even if the user inputs a lower case letter, it is easy enough to make the first letter capital when return is pressed to enter the word. This way reinforces that certain words need capitals even if you are not asking them to use them for data input.

☞ Fonts used should be clear and the younger the target audience your program is, the bigger (within reason) and bolder the letters and numbers should be. Browse around the

section of a book shop to see how the displays in the books change as the age of the child increases.

Keep control of the program simple! It's no good asking a 5 year old to press two keys on the keyboard and at the same time press fire on a joystick! Alternative choices of control are a good idea.

Sometimes the same program can be keyboard, joystick or mouse controlled.

In multi-program packages, try to give a feeling of continuity, e.g. a central theme of a circus, use the same font and colour schemes for text displays. The use of a central character in a package, will link the various parts of the program together.

Have you included alert boxes to inform the user of the choice they have made? e.g. If the child has chosen to quit the current section and return to the main menu but has pressed the wrong key, it could involve a long wait while the menu is loaded, then the game section is chosen and loaded again so that the player can continue playing. This would also mean that he/she has to start again rather than continuing from the last point as they could have if they had been given the choice of 'Are you sure?' before quitting.

If you get comments such as 'Why didn't anyone else think of that?' then you know you've got a good idea!

Listen to the things that other people say when they are viewing your work. If you use children to test, then it is important to listen to their off the cuff



remarks while they are playing as that is when they are at the guarded and their true feelings come out! Many changes have been made to our programs after listening to comments like these, if you think they think *after* the game has been played, most children will say 'fantastic!' or 'It's boring!' but will not go into details why. Listening to what they play will give you far more information. On one test session we made a video of the children playing to play back later and this was very well. The camera did not make them any less vocal as to their feelings!

General Comments.

☞ If you have put every effort into seeing that your program is the very best you can do, listened to other people's comments, included the ones that improve your program, tested it for hours and hours until you never want to see it again, it deserves to succeed, or at least get yourself noticed. It is a good spot if a programmer really *cares* about his or her work, your program will look and feel good.

☞ Never say 'It'll do!' **IT WON'T!** This attitude shows when there are lots of little errors in a program, e.g. the odd graphic that jerks or text appearing slightly out of place.

☞ If your program is returned from wherever you've sent it, with a list of comments or 'improvements' consider them before sending the same program elsewhere, if someone has taken the trouble to comment, they are probably worth taking your program with the changes.

☞ Be flexible! If you want to succeed, you must be willing to change your views if necessary in order to get a software house to accept you and your work.

☞ Don't expect too much too soon. Here we are not just talking about money! Just because you think that your program

is fantastic, it does not mean that everyone else will! Standards are continually rising and so as time goes by programs will have to get better and better to stand a chance out there. That's where we hope that this book, the PD Library and *Totally AMOS* will be able to help. We believe in passing on things we learn in order to raise the general standard of Amos programming, and we want everybody else to do the same!

Chapter 12

Graphics

Everyone is a born artist, most of us have to work hard to produce something usable while most people have to rely on the talents of others to create their game graphics.

We did not have a professional artist available to help with the graphics for this book, the ones you see are our own and should not be taken as an example of commercial quality artwork!

Hints and tips we can give you about producing graphics can only be drawn from our own experience and things we have learnt by looking at the work of other people.

Probably everyone has a version of Deluxe Paint, as this is one of the products included with a new machine. This is the one we use, although many people are just as happy with other products, even with the Public Domain art packages in the AMOS PD Library.

There are hundreds of clipart disks available from PD Libraries and they are of great use to programmers. A lot of educational programs can be written using graphics adapted from images found on clipart disks as you do not need the high calibre animations seen in arcade games. Static background screens can be just as interesting as parallax scrolling to a young child!

The best type of clipart to look out for are line drawn images such as you would find in children's colouring books. These have usually been saved in and saved to disks for PD users.

You might find it a worthwhile addition to your set-up. The hand-held colour scanners are now a reasonably priced item and you may be able to get a second hand bargain now that hand-held colour scanners are around.

You must be sure that the one you buy is suitable for your machine, otherwise you could be wasting your time and money.

If you want to use clipart as a source of graphics, then the best thing is to start collecting as many clipart disks as you can so that you have a ready supply to hand when they are needed. Look in the advertisement magazines for the best priced ones! Some libraries will offer a special price, some will have subject specific disks, some colour and some outline art. Shop around to get the type you want, but expect the person who answers the phone to be able to tell you if they have a specific picture, but they should be able to tell you the clipart they carry.

You will also occasionally find disks of bobs and backgrounds in a PD Library although these are not numerous. The Sprite which used to be given away with early copies of AMOS is now in the PD Library (APD 433). This contains loads of animated sprites to play with, but cannot be used to produce a commercial program without the permission of Europress Software.

Fonts also come under the heading of graphics, again there are disks containing .IFF fonts on PD Libraries which you can use in your program look more professional.

There is a very good PD tutorial for DPaint available called DPaint Gallery. This deals mainly with creating logos such as seen in the book and on title screens, but the effects taught could be applied to

Some Hints AND Tips

Use 16 colour palettes as much as possible as this will save you. 32 colours should be the maximum. Even if you have an Amiga, you will be restricting your selling potential if only those with an Amiga Chip set can load your program. In a couple of years time, different as the older models wear out and the new machines come out. Very many commercial games are done using only 16 colours, and

if it is set up carefully, you will be able to create some great effects.

Remember that you must NEVER use colour 0, i.e. the one at the top left of the palette in DPaint as this is seen as transparent in AMOS and could cause some strange effects on your work. Colour 0 is the colour that appears as the border around the edges of the screen.

When setting up your palette, make sure that you have at least 2 shades of each colour - look at the palette we use, it is basically the same throughout the disk with only minor colour changes to allow for differences in the games' needs.

Quick Flash!

At all possible, reserve 2 colours in your palette and make sure that you do not use them in your background picture or in any bobs. If you are using a 32 colour palette, then this should not cause any problems. You can then use these colours to form a twinkling lights effect, for example, as you will see in the reward sequence of the Duckshoot program.

In that example, the curtains at each side of the stall have been edged with alternating pixels in these 2 colours. When the Flash command is used, the lights seem to move around the borders of the curtain in an effect such as you would see in Christmas lights or outside theatres.

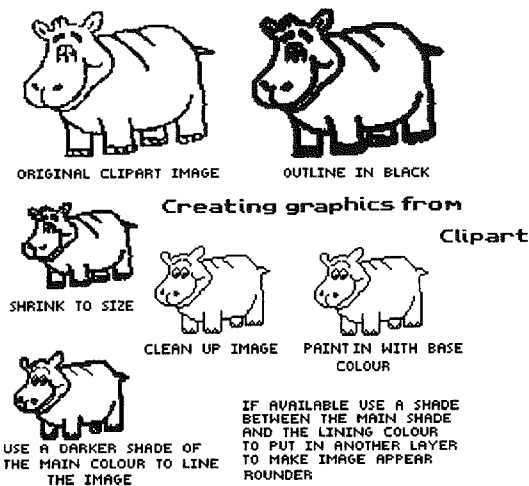
Creating Graphics from Clipart

On the disk which comes with this book you will find a screen which is a full colour version of the illustration below.

In the Crosswords game background you will see a hippo. This was found on a clipart disk and processed into a suitable form to use with the program.

It started life as part of a high-res screen full of images. We are using low resolution 16 colours so set DPaint up for this. To be sure you get the image in

an identical set up as your other art work, first load a screen that has been working on then press 'J' to switch to a blank screen. Load the clipart screen and when the requester asks if you want to change the format to the one being loaded, answer 'No'. this will give you a version of the clipart and will have destroyed your original palette. You should use the 'Restore Palette' option to get it back. Now you have a funny coloured clipart screen. To get this back to black and white use a stencil protecting the outline colour and draw a solid square over the image you want to use. Now repeat protecting the background and drawing a solid black square over the image. Save the stencil as this causes problems if left on.



the picture as a brush again and outline by pressing 'o'. This gives a very thick outlined image which will shrink a lot better than the

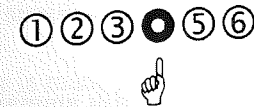
Now all you have to do is to 'clean up' the picture by colouring the pixels in white and filling gaps with black. You should now have a version of the original altered to your satisfaction and ready to

In our example we have a hippo and 3 shades of grey. Paint the

in the lightest grey, then draw in a 'lining' of the darkest grey followed by another in the middle shade. This helps to give a rounded effect to the picture. This works with just 2 shades of a colour as well.

Experimenting is the name of the game, the more you try out, the better you will get.

Painting buttons



Grab a brush. Buttons are very popular in all types of programs and give a 3D look to image and control panels.

screen to white. Buttons are easy enough to create if you follow the basic rules. Decide if the light is coming from the right or the left. We shall choose the left. For Change the screen to standard - that will be the button when it is out in the 'Off' position - draw a rectangle with the left and top edges in light shades and bottom and right edges in dark shades. For the alternate image, draw a

The image is too large to use, so have to be shrunk. The top and left and light ones at the bottom and to the right. The image as the mouse button is too thin, so pick your pen colour. This often doesn't seem necessary to create the effect.

These 2 images are then toggled in your program when the mouse button is clicked on a zone over the button.

This effect is also useful for drawing in wooden panels on doors or for making parts of an otherwise flat surface stand out from the rest.

Example:

You have a plain brown door painted in a mid brown shade. You have a dark brown and a flesh shade available in your palette.

You want to put an embossed diamond shape on the door. Draw in your

diamond with dark brown lines, leaving the middle in the original. Then redraw 2 adjoining sides in flesh. This will make the diamond stand out from the main part of the door. The light coloured sides are the ones which face your light source. Try changing the placement of the dark colours to see how the effect changes. Remember to keep your character on the same side throughout your picture or your picture will look quite right.

Animations

There is not much I can say to help with animations, as this is not one of my stronger points! However, this is how I approach an animation of a character walking from left to right.

Draw your first image e.g. a man, in the standing still position. On a separate sheet of paper draw a brush and plant it next to the first so that you will be able to see the differences. To get a decent walking action you will need to draw 4 frames. The first frame shows the head, arms and legs and possibly slight changes to the outline of the stomach and back.

When setting the number of the frames remember to calculate the total number needed for the whole animation, not just what is needed for a single part of the anim. For example, if the legs need 3 frames and the arms need 4, then the whole anim has to be spread over the maximum number of frames.

Do not attempt to animate all the parts of the body at the same time. This gets a bit confusing. Start with the legs, do all the frames needed for them. Then do the arms. 4 will be sufficient for most anims.

Paste down a copy of all the frames with the leg anims complete. Then work on each one in turn animating the arms, then the head etc.

Test each stage out in the anims function in DPaint to see if they work. Remember to slow down the frames per second rate to between 10 and 20. That way you can see the animation work at a reasonable speed. Move

to frame manually will show up problem areas quite well.

Remember to make your character the same height in the facing front view as in the side view, sounds elementary, but easily forgotten.

Simple spot animations may need only two frames. In the scrolling menu, the doors of the buildings have only 2 frames. The doors of the buildings are shown in the default backdrop and an alternate image of an open door is overlaid to simulate opening if the building is clicked on.

The animation of a girl posting the letter is made up of only 2 frames, one side view and one front view. The clock is made up of four images each of which shows a different time. The images cycle through at each mouse click. The carousel is another 'cheat' anim! The roundabout was drawn with 4 different coloured horses. There are 4 frames, frames 1 & 3 show the horses in the 'up' position and the other 2 in the 'down' position. Frames 2 & 4 show the opposite. This will give the up and down movement of the horses. To make the horses go around, move the colour of each horse to the left in each frame so that when they animate, they will appear to go around. This works reasonably well here because the bob is small.

Another trick animation can be achieved by using the AMOS Flash command. The sort of thing this works on is like that seen in a row of flashing Christmas tree lights where the lights seem to move along in a line. When, in fact alternate lights are flashing on and off in turn tricking the eye into thinking that the lights are moving along a path.

To do this, you must choose 2 colours which are used **nowhere** else in the picture, because if they are, then those bits will flash as well! Imagine you are want to create a row of lights along the edge of a circus tent. Draw in two lines to act as the edges of the rope of lights leaving a gap of a pixel or two in-between for the lights. Fill in this gap with alternate pixels of your 2 chosen flash colours Blue and white work well together to produce this effect. When you come to program these lights, you will find that the colours change from one to the other thus giving the illusion

of movement. Remember that you can create a second blue and your palette so that the rest of your picture can also contain these it is the colour numbers in the palette that the program needs.

e.g. Your palette contains 2 occurrences of royal blue and 2 white colours you have chosen as the flash colours are the last on the your DPaint palette. In a 16 colour screen these will be colours 1 and 2 and as long as these have not been used anywhere else on the screen should get the effect you are after.

Chapter 13

are to get some help...



When Amos first came out, there were very few sources of help available. This was, pretty obviously, because only a select few had had the privilege of Beta testing Amos and so were the only ones in a position to help other users.

Now, things are very different! Amos celebrated its third birthday in June 1993 and there are plenty of people who are willing and able to help with just about any Amos problem.

The Official AMOS PD Library.
1 Penmynydd Road,
Penlan,
Swansea
West Glam
SA5 7EH.
☎ 0792 588156

We have been running the Library since May 1992. Here you will find hundreds of disks containing AMOS source code which you can load into AMOS to see how different coders have tackled various AMOS problems. These programs contain code which you are free to use in your own programs, but it is only courtesy that if you do use someone else's work, that you should credit them for it in your program, this is usually done in the form of Remmed lines in the listing of the uncompiled program, or in a credits screen displayed from a compiled program.

There are two main categories of program in the library:

AMOS PUBLIC DOMAIN (APD) DISKS

These are numbered, at the time of writing, from 1 to 489. There are a few gaps as some disks were withdrawn for various reasons in the past. Most of these disks contain source code. In fact, we have now made a rule that source must be on any disk that is included in this section. If a programmer is supplying a compiled program, then the source code must be given as a .amos file on the disk, or even as an Ascii file which can be loaded into the editor.

We have 38 disks of just source code. These disks have been put together from smaller programs sent in by programmers and represent a real value for money.

The AMOS disks contain loads of versions for different types of programs: screen wipes, text programs, games - in fact just about everything you could want to do in AMOS is covered somewhere in the library. If you are stuck with a particular problem, then this could be a good way of finding out how to go about tackling the problem.

Also on the library are disks of music and sound effects. Most of the music is in the form of .abk files that can be played directly from the disk. The disks usually have a music player on them as well so all you need to do is load it to play through the tunes to see which ones you want to use.

The Sams disks contain hundreds of sound effects which you will need to make your game complete. Most programmers collect as many sound samples as possible so that they will then usually have the right sound available as it is needed.

One request from us about ordering disks of Sams or music is that it is impossible to remember the names of all the sams and the type of sound on each disk! So please do not write or phone to ask if we have

goes 'crash, bang wallop!' or a piece of music which would suit a shoot'em up! The odds are we won't know offhand and it would take hours to play through each disk to find the right one! Thanks!

GENERAL PD (GPD) DISKS

We have now reached number 220 and contain a lot of non-AMOS programs as well as AMOS programs which have no source code. Some programmers do not wish anyone else to see their code, so we respect this and put them in this section. Many of the non AMOS disks are very useful to programmers as well - there are many utilities such as disk editors and disk managers as well as Sample handling programs and other disks. Many of the Amos disks contain Demos where programmers challenge each other to push AMOS to its limits.

As well as the above disks, we have a couple of hundred disks of clip art. These can be very useful to those who cannot draw too well as they can give assistance in starting off a picture.

In the clipart collection are disks of coloured clipart from Australia which contain just about everything from flowers to composers!

Monthly AMOS

This is our bi-monthly disk magazine which we have produced since September 1991. We want to help *all* AMOS users, but we do aim for the beginners. We see TA as a meeting place for AMOS users to meet and share their triumphs and tragedies and so help each other overcome the feelings of isolation that so often swamp programmers.

We invite readers to write in with problems, reviews of AMOS programs and tutorials which will help other readers. We have a few 'regular' contributors who have supported us from the start, as well as new faces who get set on seeing that TA carries on helping others!

Issue 0 is now PD and is available as APD 341.

Issue 1 was published in November 1991 and an issue has appeared 2 months since then. Issue 12 was published in September 1992.

The Official AMOS Club

Aaron runs the Official Amos Club which offers a help line and newsletters for a subscription of £12.00p

Aaron Fothergill
1 Lower Moor,
Whiddon Valley,
Barnstaple.
N Devon.
EX32 8NW

Canada

For those Amos users in Canada, there is now a place where you can get ALL the Amos Public Domain Disks and Europress Software without spending extra time and money buying from the U.K. Jim Rhodes, Proprietor of SeaScape Software in Winnipeg and he will be able to help you with all your Amos needs.

SeaScape Software
106a-2621 Portage Avenue
Winnipeg, Manitoba
R3J-0P7

☎ CANADA (204) 889 3357



Australia



Australian Readers also have a 'local' source of Amos PD, Norm Victoria owns Allen Computer Supplies and he also stocks Amos Public Domain Disks.

Allen Computer Supplies
432 Dorset Road
Croydon
Victoria 3136
Australia

☎ +61 3 723 1780



The Mr AMOS Club



This is a relatively new Amos disk magazine produced by Brian Bell. The magazine is published bi-monthly, please send SAE or IRC to Brian for details. There are lots of articles for both beginners and more experienced users as well as plenty of code, music and graphics.

The Mr AMOS Club
8 Magnolia Park
Dunmurray
Belfast
N. Ireland
BT17 0DS

Chapter 14

Where to send your programs....

Now that you have written your program, and shown it to all your family and friends who say it's wonderful, what do you do with it?

It really depends on why it was written in the first place and what type of program it is. If it is a database which you wrote for your kid brother to catalogue all his pet spiders, then it is not going to appeal to many other people! On the other hand, if you wrote a game to help your kid brother with his maths and he enjoys using it, then it will be useful to other people who will make good use of it.

Having decided that your program could interest other people, you must decide where to send it. There are several options open to you at this stage, and also several things which you need to do to your program before you pop it into a jiffy bag and post it.

Your program should autoboot from your supplied disk so that the person who wants to see it can just 'load & go'. (this is not necessary for Amos or source code routines sent to PD - see later)

Make sure that the program works properly. This sounds silly if you've tested it a few times without it crashing, but you really need to watch someone else using it as well so that you can take a back seat and observe how easy it is to use and also if your tester does something unexpected which causes the program to crash. This happens more often than not in educational games, especially if the child using it is very young. Keys are pressed at the wrong time, mouse buttons clicked repeatedly over certain parts of the screen can cause really weird effects! You have to see a youngster in action to believe some of the things they will try doing!

Right, so it survived the testing, now you must decide where your program should go.

The main choices are:-

- Public Domain
- Shareware
- Licenseware
- Marketing your own work
- Commercial software house

We'll take each one of these in turn and explain what each one is



Public Domain



Public Domain, as relevant to software, basically means the programmer does not hold copyright to his work and it is made available to the public for them to use without restraint of legal action. It is usual though for credit to be given to the original coder if someone uses the program. Other people are not allowed to make money from the sale of the program.

Public Domain Libraries, such as The AMOS PD Library, sell disks containing these programs. The charge made covers the disk, postage and packing and time involved in copying and distributing the disks.

If you want to release your program as P.D., then all you have to do is make copies of the program and send it out to as many libraries as you want. They will then decide if they wish to include it in their catalog.

At the AMOS P.D. Library we accept programs which can only be run using a version of AMOS. This will not usually be acceptable for libraries who do not specialise in AMOS programming as they have to provide disks to autoboot. We provide disks containing only source code and collect from AMOS programmers until we have enough to fill a disk. The highest number of programs on a disk to date is 119! Obviously not all mega game, but routines such as screen wipes, music player routines, bob routines etc. If you have kept loads of small programs

we have written whilst learning to program, and do not mind sharing what you have learnt with others then send them in to us. Disks like this are greatly sought after by other programmers, most of whom are beginners needing to find out how to tackle a problem.

If your program is a stand-alone program then Compile or Ramos it so that it autoboots. If you are sending it to us, and want it to be considered for the APD section (AMOS PD) then, if Compiled, please include the source code as this is our rule! This can be sent on the disk as a .amos file or saved out as Ascii which can then be merged into Amos' editor.

If you want to keep your code hidden, then send a Compiled version and your wish will be respected and your disk entered into the GPD (General Public Domain) section.



Shareware



This is a slightly different version of the PD idea, but where you might earn a bit of money!

Shareware means that your disk is sold as PD, but you must include a message on the disk to say that the program is Shareware and that if they find the program useful, then they should send a registration fee to you to register their copy as legal.

Usually the version of the program sent as PD is incomplete in some way, e.g. In a game, only the first few levels are available so that if the user likes it and wants the complete program, then he must pay you for the full version - the fee charged usually reflects the type of program and the level of 'after-sales service' offered.

The registration fee is up to you, but overpriced fees with little extra to offer the user is more likely to fail than a reasonable fee with the offer of the full game plus any update to it.

You must remember that Shareware on the Amiga in the UK does not

have a very good reputation unless you are Jeff Minter who has made a nice amount from his games - he gives a good deal for his fees so is seen as fair. Shareware on the PC in the USA can make you rich - but do not reserve a new car on the strength of Shareware fees here before you see the response.



Licenseware



This is another progression from the idea of PD and Shareware.

Licenseware schemes are run by some PD Libraries in the UK who pay the programmer a set royalty fee for each copy of a program that they sell for him or her.

This scheme can work very well for most people, and can earn a usable amount of money - especially if you have several titles released into the scheme. It does depend on the honesty of the distributors, and you will sign an agreement that it can only be sold by the distributor of the scheme i.e. you cannot advertise them and sell your titles yourself unless you also become a distributor. For their part, the distributor advertises your titles and does their part in sending reviewers copies etc.

Acceptance onto a licenceware scheme is NOT automatic, you have to submit your program to the organisers who will test it and see if they think it will sell well enough to pay for their costs. This is a gentle introduction into commercial work inasmuch as you are asking someone if your work is good enough to sell. The standards are higher than those set for P.D. and the copyright remains with the programmer, so if you are accepted onto the scheme, then no-one else can steal your idea. These programs should be compiled as an added way of protecting your work.



Marketing Your Own Work



This is another option to be considered if you want to keep all the profits from the sales of the disks. There are a lot of things that you will have to

consider before you go ahead.

To let people know that your program exists, you will have to advertise. This can be very costly if you use the national computer magazines, but they do reach thousands of readers. You will probably need to spend over £300 per month for 1/8 of a page, so you will have to have some financial backing to follow this route.

You could send off copies off for review to magazines, and if your program is chosen (remember to include all details of how and where to get it!) you should get some response, especially if the reviewer liked the program.

Our magazine, 'Totally AMOS' allows free advert for subscribers, but although we have a good following, it does not run into thousands, so you should consider contacting as many clubs or user groups as possible about advertising.

Some mail order PD Libraries might be willing to put small printed adverts in with all orders they send out. These inserts should be kept small as even 1 extra A4 sheet can push up the postal cost to the people sending out the mail, and this might be passed on to you.

Sending out demo versions to PD Libraries will advertise your program as you can put all the information necessary in the program. This can be done in the same way as Shareware.

Next you will have to think of what you will need to actually send out your disks. The most obvious is a source of reliable disks. A batch of faulty disks will cost more in the long term as you will have to replace any which you send out to your customers at your own cost.

The disk should be tidily labelled - if you are charging for the disk, then your customer will expect to see something which will look worth paying for. It is not necessary to provide coloured printed labels made by a professional printer as these come in large quantities, usually a minimum

of at least 1000, and are expensive to set up. If you are charging for example £10 for the disk, then this might be worth thinking of as it will really make the disk look great. We started off by writing out labels for *Totally* AMOS by hand, but as soon as the response warranted the expense, we had special labels printed and from remarks made by our readers it was well worth it! A cheaper alternative is to print the labels yourself - the quality of these will obviously depend on the printer you are using.

Next, you have to get the disks to your customers in one piece. This means that you will have to use padded envelopes to protect the disks from the rigors of the mailing system. These envelopes are best bought in bulk from an office stationery outlet when the numbers of orders warrant the outlay.

Finally, you must consider the time involved in sending out disks. If you are only marketing a few titles, then you can go ahead and copy in advance of orders so that all you will have to do is pop the disks into an envelope, write the address and stick on a stamp.

If you are thinking of marketing your own titles, take into account how much free time you can give to the idea. If you have to spend the day at work, school or college etc. then you will not have enough time to cope with the work involved if your title really takes off. When planning doing this, imagine it at its worst - very few orders - and also at its best - so many orders that you do not have the time to get them out. Plan on getting help just in case you cannot cope, that way you will not have to disappoint your customers.

Remember, that by marketing your own disks you are receiving an income and so will be eligible for all the officialdom that goes with it - i.e. the Tax Man and maybe the VAT people as well. Check up with your local tax office to find out more.



Commercial Releases



This is the most difficult area to get into for obvious reasons. You do not need a University degree to get a job with a software house, lots of programmers have only ever produced usable programs with AMOS - and that includes us!

If you think that your program has real commercial possibilities, then there is nothing to stop you from contacting a software house, some of them advertise in the computer magazines for programmers to do just that. You must be patient as larger companies can take a long time to reply.

You should at least have summary of the game specs ready before you contact a company so that you will be able to tell them about your idea without giving the whole idea over on the first contact. You may, of course, have the game or program already finished, but at this stage you should only send a demo version just to show prospective publishers how the program will run. Include written documentation for them to read as well, a brief outline of what is included in the demo version, what else is in the full program and instructions on how to use the program.

It can be agonising waiting for replies to these submissions, and there will probably be a lot of disappointments along the way, but keep on trying, if your program has commercial potential, then someone out there will want it - try all avenues until you find the right place for your work.

This book is not written for professional coders, but for enthusiastic learners who (probably) want to earn money from their hobby, so we advise you all to take things a step at a time and progress from one stage to another. It's great to have ambition, but to be given a commercial project you will need to have a coding 'history'. This means that the people you work for will want to see what other things you have produced whether on your own or as part of a team. This is where a few Licenseware titles are very useful, especially if they have been successful

as they will show that you have the talent to produce a product which sells.

Also by sending out your work as PD, Shareware or Licenseware, you will find that your name will become known and who knows who might get in touch with you!

We, at the AMOS PD Library, look at all the programs that are sent in and will not automatically put a program into the PD section if we believe that it is too good, although it means lost sales for us. If we think that it is good enough to be considered for Licenseware, we will write and tell the programmer so that it can be sent to the right person, we cannot, however, guarantee that it will be accepted for Licenseware, but at least it will have had a chance.

Very occasionally we receive a disk which is just too good for anything but a commercial future and we have contacts in commercial houses who are always on the look out for new talent. It is not only the standard of coding which is important - maybe you have a future as an artist or musician, or it might be the game idea itself which gets you noticed. Good graphics obviously will help a program, but the idea of the game can be more important to a software house who have access to professional artists and musicians. Some will insist that the graphics and sounds are redone by their own people anyway.

Just keep on trying and practising, learn new things, there's more that can be done with AMOS than you would imagine!

Most importantly of all, have



Chapter 15

Useful Procedures and other Routines

One of the best ways of learning any computer language is by finding printed listings and keying them in for yourself. Those who had computers way back in the mid eighties will probably remember that the computer magazines were full of listings to be typed in. There were no cover disks in those days and so if you wanted a game or a program from the magazine, there was no alternative to keying it in for yourself. We spent hour upon hour keying listings into our Dragon 32 and we learnt a lot about how Basic worked.

These days people have listings on disk and so have become lazy, this is your chance to remedy that! The following are listings of useful bits and pieces from the PD Library - The programmers are credited if we know who wrote them, for those which remain anonymous, thanks.

As usual with listings, the AMOS program lines are longer than can be fitted onto a single line on a page, so if you see ↵ at the end of a line, then do not press Return, but continue on the same line in the AMOS editor.

Listing 1

```
Do
PLASMA1
PLASMA2
PLASMA3
Loop
Procedure PLASMA1
Screen Open 0, 480, 400, 32, Lowres
Curs Off
Flash Off
```



```

Cls 0
Screen Hide 0
Palette $F,$11F,$22F,$33F,$44F,$55F,$66F, ↵
$77F,$88F,$99F,$AAF,$BBF,$CCF,$DDF,$EEF,$FFF, ↵
$EEF,$DDF,$CCF,$BBF,$AAF,$99F,$88F,$77F,$66F, ↵
$55F,$44F,$33F,$22F,$11F,$F,$E
CL=1
CLF=1
For S=1 To 600 Step 2
Ink CL
Draw 0,S To 480,S
Draw 0,S+1 To 480,S+1
If CLF=1 Then Inc CL : If CL>31 Then CL=0
Next S
Screen Open 1,480,400,32,Lowres
Curs Off
Flash Off
Cls 0
Get Palette 0
Screen Hide 1
Degree
M=20
BB=1
For S=1 To 480
Screen Copy 0,S,1,S+1,350 To 1,S,M*Sin(S)-40
If BB=10 Then Inc M : BB=1
Next S
Screen Close 0
Screen Hide 1
Screen Open 0,480,400,32,Lowres
Curs Off
Flash Off
Cls 0
Palette $0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0, ↵
$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0, ↵
$0,$0,$0,$0,$0,$0,$0

```

```

Fade 5 To 1
For S=1 To 300 Step 1
Screen Copy 1,1,S,480,S+1 To 0,(47*Sin*(S*4)-70),S
Next S
Screen Display 0,100,40,460,360
Channel 1 To Screen Offset 0
A$="L: LY=1;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+2;P;"
A$=A$+"LY=Y+2;P;LY=Y+2;P;LY=Y+3;P;"
A$=A$+"LY=Y+3;P;LY=Y+3;P;LY=Y+3;P;"
A$=A$+"LY=Y+2;P;LY=Y+2;P;LY=Y+2;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-2;P;LY=Y-2;P;LY=Y-2;P;"
A$=A$+"LY=Y-3;P;LY=Y-3;P;LY=Y-3;P;"
A$=A$+"LY=Y-3;P;LY=Y-3;P;LY=Y-2;P;"
A$=A$+"LY=Y-2;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;P;JL"
Amal 1,A$
Screen Close 1
Shift Up 1,0,31,1
Amal On
TIM=1
Repeat
Inc TIM
Wait Vbl
Until TIM>500
Fade 1
Wait 20
Screen Close 0
End Proc
Procedure PLASMA2
Screen Open 0,480,400,32,Lowres

```



```

Curs Off
Flash Off
Cls 0
Palette $F00,$F22,$F44,$F66,$F88,$FAA, ↵
$FCC,$FEE,$FFF,$EEF,$CCF,$AAF,$88F,$66F,$44F, ↵
$22F,$22E,$22C,$22A,$228,$226,$224,$222,$400, ↵
$600,$800,$A00,$B00,$C00,$D00,$E00,$F00
Screen Hide 0
CL=1
CLF=1
For S=1 To 600 Step 2
Ink CL
Draw 0,S To 480,S
Draw 0,S+1 To 480,S+1
If CLF=1 Then Inc CL : If CL>31 Then CL=0
Next S
Screen Open 1,480,400,32,Lowres
Curs Off
Flash Off
Cls 0
Get Palette 0
Screen Hide 1
Degree
M=20
BB=1
For S=1 To 480
Screen Copy 0,S,1,S+1,350 To* 1,S,M*Sin(S*2)-40
Next S
Screen Close 0
Screen Hide 1
Screen Open 0,480,400,32,Lowres
Curs Off
Flash Off
Cls 0
Palette $0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0, ↵
$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0, ↵

```

```

$0,$0,$0,$0,$0,$0,$0,$0
Fade 5 To 1
For S=1 To 300 Step 2
Screen Copy 1,1,S,480,S+1 To 0,(47*Sin(S*4)-70),S
Screen Copy 1,1,S+1,480,S+2 To ↵
0,(17*Cos(S*4)-70),S+1
Next S
Screen Display 0,100,40,460,360
Channel 1 To Screen Offset 0
A$="L: LY=1;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+2;P;"
A$=A$+"LY=Y+2;P;LY=Y+2;P;LY=Y+3;P;"
A$=A$+"LY=Y+3;P;LY=Y+3;P;LY=Y+3;P;"
A$=A$+"LY=Y+2;P;LY=Y+2;P;LY=Y+2;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y+1;P;LY=Y+1;P;LY=Y+1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-2;P;LY=Y-2;P;LY=Y-2;P;"
A$=A$+"LY=Y-3;P;LY=Y-3;P;LY=Y-3;P;"
A$=A$+"LY=Y-3;P;LY=Y-3;P;LY=Y-2;P;"
A$=A$+"LY=Y-2;P;LY=Y-1;P;LY=Y-1;P;"
A$=A$+"LY=Y-1;P;LY=Y-1;P;P;JL"
Amal 1,A$
Screen Close 1
Shift Up 1,0,31,1
Amal On
TIM=1
Repeat
Inc TIM
Wait Vbl
Until TIM>500
Fade 1
Wait 20
Screen Close 0

```



```

End Proc
Procedure PLASMA3
Screen Open 0, 480, 400, 32, Lowres
Curs Off
Flash Off
Cls 0
Palette $F0F, $F2F, $F4F, $F6F, $F2F, $F4F,
$F6F, $F8F, $FAF, $EBF, $CCF, $ADF, $8EF, $6FF, $4EF,
$2DF, $2CE, $2BC, $2AA, $298, $286, $274, $262, $450,
$640, $830, $A20, $B10, $C00, $D00, $E00, $F00
Screen Hide 0
CL=1
CLF=1
For S=1 To 600 Step 2
Ink CL
Draw 0, S To 480, S
Draw 0, S+1 To 480, S+1
If CLF=1 Then Inc CL : If CL>31 Then CL=0
Next S
Screen Open 1, 480, 400, 32, Lowres
Curs Off
Flash Off
Cls 0
Get Palette 0
Screen Hide 1
Degree
M=30
BB=1
For S=1 To 480
Screen Copy 0, S, 1, S+1, 350 To 1, S, M*Sin(S*4)-40
Next S
Screen Close 0
Screen Hide 1
Screen Open 0, 480, 400, 32, Lowres
Curs Off
Flash Off

```

```

Cls 0
Palette $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0,
$0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0,
$0, $0, $0, $0, $0, $0
Fade 5 To 1
For S=1 To 300 Step 2
Screen Copy 1, 1, S, 480, S+1 To 0, (17*Sin(S*4)-70), S
Screen Copy 1, 1, S+1, 480, S+2 To
0, (17*Cos(S*4)-70), S+1
Next S
Screen Display 0, 100, 40, 460, 360
Screen Close 1
Shift Up 1, 0, 31, 1
TIM=1
Repeat
Inc TIM
Wait Vbl
Until TIM>500
Fade 1
Wait 20
Screen Close 0
End Proc

```

Listing 2

```

Hide On
Screen Open 0, 360, 270, 32, Lowres
Curs Off
Flash Off
Cls 0
Palette $0, $99F, $0, $0, $0, $0, $0, $0, $0, $0, $0,
$0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0, $0,
$0, $0, $0, $0, $0, $0
Dim STCL(150)
Dim STY(150)
For A=1 To 150

```



```

STCL(A)=1+Rnd(31)
STY(A)=1+Rnd(270)
Next A
Shift Up 1,1,31,1
For A=1 To 360 Step 5
For B=1 To 40
Ink STCL(B)
Plot A,STY(B)
Inc STCL(B) : If STCL(B)>31 Then STCL(B)=1
Next B
Next A
For A=1 To 360 Step 8
For B=41 To 60
Ink STCL(B)
Plot A,STY(B)
Inc STCL(B) : If STCL(B)>31 Then STCL(B)=1
Next B
Next A
For A=1 To 360 Step 9
For B=61 To 100
Ink STCL(B)
Plot A,STY(B)
Inc STCL(B) : If STCL(B)>31 Then STCL(B)=1
Next B
Next A
For A=1 To 360 Step 10
For B=100 To 150
Ink STCL(B)
Plot A,STY(B)
Inc STCL(B) : If STCL(B)>31 Then STCL(B)=1
Next B
Next A
Wait Key

```

Listing 3

```

Screen Open 0,340,20,2,Lowres
Curs Off
Flash 1,"(111,2)(333,2)(555,2)(777,2)(999,2) ↵
(777,2)(555,2)(333,2)(111,2)"
Palette $0,$FFF
Cls 0
Set Rainbow 0,0,60,"","","(5,1,15)"
Rainbow 0,0,290,60
Channel 2 To Rainbow 0
Amal2,"FR1=1 T 50;LY=Y-1;P;NR1 "
Amal On
Def Scroll 1,1,1 To 340,20,-1,0
Def Scroll 2,50,1 To 270,20,-1,0
Def Scroll 3,100,1 To 220,20,-1,0
Def Scroll 4,150,1 To 170,20,-1,0
Screen Display 0,,160,320,
Channel 1 To Screen Display 0
Amal1,"L:LY=140;FR1=1 T ↵
15;LY=Y+R1;NR1;LR1=15;FR2=1 T ↵
15;LY=Y-R1;LR1=R1-1;NR2;JL"
Amal On
T$="WIERD SCROLLER, EH MATES??? HE HE HE!! ↵
THE PRODUCT OF A DERANGED MIND! "
P=1
CN=1
Repeat
Inc CN : If CN>9 Then CN=1 : Gosub NEWLET
Scroll 1
Scroll 2
Scroll 3
Scroll 4
Wait Vbl
Until Mouse Key
NEWLET:

```



```

Locate 40,1
Print Mid$(T$,P,1);
Inc P : If P>Len(T$) Then P=1
Return

```

Listing 4

```

Degree
Screen Open 0,320,200,4,Lowres
Curs Off
Flash Off
Cls 0
Set Rainbow 0,1,200,"(40,1,15)", "(5,1,15) (5,- ↵
1,15)", "(10,1,15) (10,-1,15)"
Rainbow 0,0,40,200
Double Buffer
Autoback 0
CNT=1
STP=6
JUMP=20
SIZE=80
SF=1
Do
Ink 1
For A=1 To 320 Step JUMP
Draw A,SIZE*Sin(CNT+A)+100 To ↵
A+JUMP,SIZE*Sin(CNT+JUMP+A)+100
Next A
Ink 0
CNT=CNT+STP
Screen Swap
If SF=1 Then Inc SIZE : If SIZE>90 Then SF=2
If SF=2 Then Dec SIZE : If SIZE<-90 Then SF=1
Wait Vbl
Cls 0
Loop

```

Listing 5

```

Screen Open 1,320,50,2,Lowres
Curs Off
Flash Off
Cls 0
Screen Display 1,,240,,
Centre "INSTRUCTIONS!"
Print
Centre "JOYSTICK L/R: DEC/INC X-SINE"
Print
Centre "JOYSTICK D/U: DEC/INC Y-SINE"
Print
Centre "L MOUSE: INC DOTS R MOUSE: DEC DOTS"
Print
Centre "BOTH BUTTONS:QUIT"
Print
Centre "JOY FIRE-RESET VALUES"
Degree
Screen Open 0,320,200,4,Lowres
Curs Off
Flash Off
Cls 0
Set Rainbow 0,1,280,"(40,1,15)", "(5,1,15) ↵
(5,-1,15)", "(10,1,15) (10,-1,15)"
Rainbow 0,0,40,280
Double Buffer
Autoback 0
Screen To Front 1
CNT=1
STP=6
JUMP=20
SIZE=0
SIZEY=0
SF=1
LI=1

```



```

Repeat
K=888
K=Joy(1)
Ink 1
For A=1 To 320 Step JUMP
PlotA+(SIZE*Cos(CNT+A)),SIZEY*Sin(CNT+A)+100
Next A
Ink 0
CNT=CNT+STP
Screen Swap
If K=1 and SIZEY>-40 Then Dec SIZEY
If K=2 and SIZEY<40 Then Inc SIZEY
If K=4 and SIZE>-80 Then Dec SIZE
If K=8 and SIZE<80 Then Inc SIZE
If K=16 Then SIZEY=0 : SIZE=0 : JUMP=20
If Mouse Key=1 Then If JUMP>10 Then Inc JUMP
If Mouse Key=2 Then If JUMP<100 Then Dec JUMP
If JUMP<=10 Then JUMP=11
If JUMP>=100 Then JUMP=99
Wait Vbl
Cls 0
Until Mouse Key=1+2

```

Listing 6

```

' OPEN SOURCE IMAGE SCREEN
Screen Open 0,320,150,2,Lowres
Curs Off
Flash Off
Cls 0
Screen Hide 0
' MESSAGE TO DISPLAY
Pen 1
Paper 0
For A=1 To 17
Print : Centre "HA HA THIS IS GOOD"

```

```

Next A
Wait Vbl
' OPEN DESTINATION SCREEN
Screen Open 1,320,150,2,Lowres
Curs Off
Flash Off
Cls 0
Screen Display 1,,100,,
Screen 1
Double Buffer
Autoback 0
' COPPERS!!
Set Rainbow 0,0,30,"","", "(1,1,15) (1,-1,15) "
Rainbow 0,0,70,30
Set Rainbow 1,0,30,"","", "(1,1,15) (1,-1,15) "
Rainbow 1,0,250,30
Set Rainbow3,1,150,"(9,1,15) (9,-1,15) ",
"(1,1,15) (5,-1,15) ", "(15,1,15) (1,-1,15) "
Rainbow 3,0,100,150
' THE WIBBLE ROUTINE!!!
SIZE=250
COUNT=1
Degree
Repeat
For A=0 To 150 Step 8
Screen Copy 0,1,A,320,A+8 To 1,SIZE*Sin(A+COUNT),A
Next A
COUNT=COUNT+5
SIZE=SIZE-1
Screen Swap
Wait Vbl
Cls 0
Until SIZE=1

```


Listing 7

```

Screen Open 0,720,250,2,Lowres
Curs Off
Flash Off
Cls 0
C=1
Palette,$F00,,,,,,,,$F00
Degree
For A=1 To 720 Step 16
For B=15 To 240 Step 16
Draw 20*Sin(A)+A,10*Cos(A+B)+B To ↵
20*Sin(A+16)+A+16,10*Cos(A+16+B)+B
Next B
Next A
Screen Open 1,320,720,2,Lowres
Curs Off
Flash Off
Cls 0
For A=1 To 720 Step 16
For B=25 To 315 Step 16
Draw B+(20*Sin(B+A)),A+(20*Cos(B+A)) ↵
To B+(20*Sin((B+16)+A)),A+16+(20*Cos((B+16)+A))
Next B
Next A
Screen Display 0,,40,320,250
Screen Display 1,,40,320,250
Wait Vbl
Channel 1 To Screen Offset 0
Amal 1,"LX=2;L:FR1=1 T 46;LX=X+8;NR1;LX=1;JL"
Channel 2 To Screen Offset 1
Amal 2,"LX=1;L:FR1=1 T 46;LY=Y+8;NR1;LY=1;JL"
Dual Playfield 0,1
Wait Vbl
Amal On
Wait Key

```

Listing 8

```

'*****
'* Sierpinski fractal coded by Shockwave *
'*****

Screen Open 0,640,230,2,Hires
Curs Off
Flash Off
Cls 0
Palette $0,$F00
Dim X(2)
Dim Y(2)
X(0)=320
Y(0)=10
X(1)=50
Y(1)=190
X(2)=590
Y(2)=190
CNR=(Rnd(2))
PX=X(CNR)
PY=Y(CNR)
PPX=PX
PPY=PY
Hide On
Screen Open 1,640,10,2,Hires
Curs Off
Flash Off
Cls 0
Screen Display 1,,280,620,
Def Scroll 1,0,0 To 640,10,-1,0
Palette,$FFF
SCRL$="Hi folks!!! just some more junk!!!"
LL=Len(SCRL$)
PS=1
PLTS=5
CC=1

```



```

Gosub COPP
Repeat
Gosub FRACPLOT
Inc CC
Screen 1
Scroll 1
If CC>8 Then Gosub NLT
Wait Vbl
Until Mouse Key
End
FRACPLOT:
Screen 0
Ink 1
For A=1 To PLTS
C=1+Rnd(7)
CNR=(Rnd(2))
PX=PX+(X(CNR)-PX)/2
PY=PY+(Y(CNR)-PY)/2
Plot PX,PY
Next A
Return
NLT:
Locate 78,0
Print Mid$(SCRL$,PS,1);
Inc PS : If PS>LL Then PS=1
CC=1
Return
COPP:
Set Rainbow 0,0,30,"(1,1,15)(1,-1,15)",
"(1,1,15)(1,-1,15)","(1,1,15)(1,-1,15)"
Rainbow 0,0,45,30
Channel 1 To Rainbow 0
Amal1,"L:LR2=1;FR1=1 T 20;LY=Y+R2;LR2=R2+1;
NR1;P;LR2=20;FR1=1 T 20;LY=Y-R2;LR2=R2-;NR;P;JL"
Amal On
Return

```

Listing 9

```

Screen Open 0,720,260,2,Lowres
Curs Off
Flash Off
Cls 0
Palette $3,$FFF
Degree
For A=10 To 720 Step 10
For B=10 To 240 Step 10
Plot 4*Sin(A+B/1.4)+A,2*Cos(A+B)+B
Next B
Next A
Channel 1 To Screen Offset 0
Amal1,"L:FR1=1 T 37;LX=X+10;NR1;LX=1;JL"
Amal On
Wait Key

```

Listing 10

```

'*****
'**** Wierd Copper Effect Coded By Shockwave
'*****
Screen Close 0
TOP=150
STRETCH1=100
STRETCH2=250
Repeat
If STRETCH1>=1 Then Gosub SIZE
Wait Vbl
Until STRETCH1=0
Wait 180
JM=TOP
For A=1 To 30
Rainbow 0,0,JM,(STRETCH1+STRETCH2)
Add JM,A : Wait Vbl

```



```

Next A
Rainbow Del 0
End
SIZE:
A$=" (" + Str$(STRETCH1) + ", 1, 15) ↵
(" + Str$(STRETCH2) + ", -1, 15) "
Set Rainbow 0, 0, 300, "", "", A$
Rainbow 0, 0, TOP, (STRETCH1 + STRETCH2)
Dec STRETCH1
Dec TOP
Return

```

Listing 11

```

Screen Open 0, 360, 240, 4, Lowres
Curs Off
Flash Off
Cls 0
Palette, $F35, $35F, $400
Cls 2
Ink 1
Polygon 0, 0 To 360, 0 To 360, 125 To 0, 125 To 0, 0
Double Buffer
Autoback 0
Degree
Repeat
Ink 1
Polygon 0, 70 To 360, 70 To 360, 125 + 50 * Cos(A + 90) To ↵
0, 125 + 50 * Sin(A) To 0, 70
Ink 3
Draw 0, 125 + 50 * Sin(A) To 360, 125 + 50 * Cos(A + 90)
Add A, 5
Screen Swap
Wait Vbl
Cls 2, 0, 80 To 360, 175
Until Fire(0)

```

Listing 12

```

'*****
' * Sine Lines (c) 1993 Shockwave of Fanatix ***
' * If you use 'em, remember to credit me! ***
'*****

Proc LINES
Procedure LINES
Screen Open 0, 320, 250, 8, Lowres
Curs Off
Flash Off
Cls 0
Degree
XOFF=150 : Rem xcentre
YOFF=100 : Rem ycentre
XMULTIPLY=100 : Rem xsinsize
YMULTIPLY=100 : Rem ysinsize
AD=60 : Rem sine gap size
D1=2 : Rem 1st sine multiplication
D2=1 : Rem 2nd sine multiplication
DEL=50 : Rem number of lines
PL=1 : Rem Palette number (1 To 3)
Add DEL, DEL
If PL=1 Then Palette$200, $200, $200, $300, ↵
$400, $500, $600, $700
If PL=2 Then
Palette$20, $20, $20, $30, *$40, $50, $60, $70
If PL=3 Then Palette$2, $112, $2, $3, $4, $5, $6, $7
IN=1
Shift Up 1, 1, 7, 1
Repeat
Add A, 2
Ink IN
Draw ↵
XMULTIPLY * Cos(A * D2) + XOFF, YMULTIPLY * Sin(A * D1) ↵
+ YOFF To XMULTIPLY * Cos((A + AD) * D2) + XOFF, ↵

```



```

YMULTIPLY*Sin((A+AD)*D1)+YOFF
Ink 0
Draw XMULTIPLY*Cos((A-DEL)*D2)+XOFF ↵
, YMULTIPLY*Sin((A-DEL)*D1)+YOFF ↵
To XMULTIPLY*Cos(((A-DEL)+AD)*D2)+XOFF, ↵
YMULTIPLY*Sin(((A-DEL)+AD)*D1)+YOFF
Inc IN : If IN=8 Then IN=1
Wait Vbl
Until Fire(0)
End Proc

```

Listing 13

```

'*** DISKINFO by Volker Stepprath 9/7/92 ***
Do
Input "Device.....: ";DEV$
If DEV$="" Then End
Dir$=DEV$ : DEV$=Dir$
INFO$=Space$(40)
DEV=Varptr(DEV$)
IN=Varptr(INFO$)
Dreg(1)=DEV
Dreg(2)=-2
LOCK=Doscall(-84)
Dreg(1)=LOCK
Dreg(2)=IN
D=Doscall(-114)
UNLOCK=Doscall(-90)
Print "Name.....: ";DEV$
Print "Errors.....: ";Leek(IN)
Print "Drive.....: ";Leek(IN+4)
N=Leek(IN+8)
If N=80 Then N$="Read only"
If N=81 Then N$="Validating"
If N=82 Then N$="Read/Write"
Print "Status.....: ";N;" ( "+N$+" )"

```

```

Print "Blocks.....: ";Leek(IN+12)
Print "Blocks used: ";Leek(IN+16)
Print "Bytes/Block: ";Leek(IN+20)
Print "Disktype....: ";
For I=24 To 27
If Peek(IN+I) <> 0: Print Chr$(Peek(IN+I));: End If
Next I
Print : Print "Disknode....: ";Leek(IN+28)
Print String$("-",30)
Loop

```

Listing 14

```

CHKPRINTER
Procedure CHKPRINTER
H=Peek($BFD000)
H$=Hex$(H)
If H$="$FF" or (H$="$FB") Then Print "Paper is out"
If H$="$FC" Then Print "Printer is on line"
If H$="$FD" or (H$="$F9") Then Print "Printer is ↵
off line"
If H$="$FE" Then Print "Printer is not on"
End Proc

```

Listing 15

```

Screen Open 0,640,200,2,Hires:Colour 1,$FFF
Flash Off
Global ICNBASE
OPEN_LIBRARY["icon.library",0]
ICNBASE=Param
If ICNBASE=0
Print "icon.library failed to open"
End
End If
'issue getdiskobject to read the icon

```



```

DISKOBJECT["dh0:cando/cando"]
DISKOBJ=Param
If DISKOBJ=0
Print "getdiskobject failed"
KLOSE_LIBRARY[ICNBASE]
End
End If
TLTYPE=Leek(DISKOBJ+54)
Repeat
TLTYPE$=""
N=0
TL=Leek(TLTYPE)
While Peek(TL+N)<>0
TLTYPE$=TLTYPE$+Chr$(Peek(TL+N))
Inc N
Wend
Print TLTYPE$
Add TLTYPE, 4
Until Len(TLTYPE$)=0
Procedure DISKOBJECT[OBJECTNAME$]
OBJECTNAME$=OBJECTNAME$+Chr$(0)
OFFSET=-78
GLUE$=Chr$($2C)+Chr$($53)+Chr$($4E)+Chr$($AE)
GLUE$=GLUE$+Chr$(Peek(Varptr(OFFSET)+2))
GLUE$=GLUE$+Chr$(Peek(Varptr(OFFSET)+3))
GLUE$=GLUE$+Chr$($4E)+Chr$($75)
Areg(0)=Varptr(OBJECTNAME$)
Call Varptr(GLUE$), ICNBASE
End Proc[Dreg(0)]
Procedure OPEN_LIBRARY[LIB$,VERS]
LIB$=LIB$+Chr$(0)
Dreg(0)=0
Areg(1)=Varptr(LIB$)
End Proc[Execall(-552)]
Procedure KLOSE_LIBRARY[LIB]
Areg(1)=LIB

```

```

R=Execall(-414)
End Proc

```

Listing 16

```

' handy tips
'To check if led is on/off
'A=Btst(1,$BFE001)
'If A=0 LED IS On
'if a=-1 led is off

```

Listing 17

```

' is there a disk in the drive
'Poke $BFD100,%10000 : Rem df0: turn drive on
briefly
'Poke $BFD100,%1000 : Rem df1: turn drive on briefly
'A=Btst(2,$BFE001)
'if a=-1 disk in drive
' if a=0 no disk in drive

```

Listing 18

```

' is the disk write protected
'Poke $BFD100,%10000 :Rem df0:turn drive on briefly
'Poke $BFD100,%1000 : Rem df1: turn drive on briefly
'A=Btst(3,$BFE001)
' if a=-1 means write enabled
' if a=0 means write protected

```

Listing 19

```

'* ©1992 by Volker Stepprath *
Amos To Back
Wait 50
Areg(0)=0

```



```

DISPLAYBEEP=Intcall(-96)
Wait 50
Amos To Front
Edit

```

Listing 20

```

'*  ©1992 by Volker Stepprath  *
Amos To Back
Do
If Not Btst(6,$BFE001) Then Print "left  ↵
mousebutton"
Exit
If Not Btst(10,$DFF016) Then Print "right ↵
mousebutton"
Exit
Loop
Amos To Front
Wait 50
Edit

```

Listing 21

```

'*  ©1992 by Volker Stepprath  *
'* Reads datas which was given in a CLI-Window
`* in a buffer *
Amos To Back
_READ
Amos To Front
Print "You feel: ";T$
End
Procedure _READ
'**** Define variables ****
Shared T$
T$="How do you feel: "+Chr$(0)
FENSTER$="CON:0/0/640/200/Moin,Moin!"+Chr$(0)

```

```

'**** Open window ****
Dreg(1)=Varptr(FENSTER$)
Dreg(2)=1006
HANDLE=Doscall(-30)
'**** Write text ****
Dreg(1)=HANDLE
Dreg(2)=Varptr(T$)
Dreg(3)=Len(T$)
XWRITE=Doscall(-48)
'**** Read text from keyboard ****
T$=Space$(80)+Chr$(0)
Dreg(1)=HANDLE
Dreg(2)=Varptr(T$)
Dreg(3)=Len(T$)
XREAD=Doscall(-42)
'**** Close window ****
Dreg(1)=HANDLE
XCLOSE=Doscall(-36)
End Proc

```

Listing 22

```

'*  ©1992 by Volker Stepprath  *
'* Writes any text in a CLI-Fenster  *
'* Actual CLI-Window ... FENSTER$=" "+Chr$(0) * ↵
Amos To Back
'**** Textstyle ****
T$=Chr$(9B)+"0;3;4;7;32;43m"
'**** Define variables ****
T$=T$+"Moin! Moin!"+Chr$(10)+Chr$(0)
FENSTER$="CON:0/0/640/200/AMOS & CLI"+Chr$(0)
'**** Open window ****
Dreg(1)=Varptr(FENSTER$)
Dreg(2)=1006
HANDLE=Doscall(-30)
'**** Write text ****

```



```

Dreg(1)=HANDLE
Dreg(2)=Varptr(T$)
Dreg(3)=Len(T$)
XWRITE=Doscall(-48)
Wait 150
'**** Close window ****
Dreg(1)=HANDLE
XCLOSE=Doscall(-36)
Amos To Front
Edit

```

Listing 23

```

'*      CLIWindow.AMOS      *
'*  ©1992 by Volker Steprath *
'*  Opens and closes a DOS-Window *
Amos To Back
'**** Define DOS-Window ****
FENSTER$="CON:10/10/600/180/DOS-Window"+Chr$(0)
'**** Open DOS-Window ****
Dreg(1)=Varptr(FENSTER$)
Dreg(2)=1006
HANDLE=Doscall(-30)
Wait 200
'**** Close DOS-Window ****
Dreg(1)=HANDLE
XCLOSE=Doscall(-36)
Amos To Front
Edit

```

Listing 24

```

'*      Comment.AMOS      *
'*  ©1992 by Volker Steprath *
'*  Set the comment of any file *
COMMENT$="Moin! Moin!" + Chr$(0)

```

```

'**** Choose file ****
FILE$=Fsel$("", "", "Please choose file", ↵
"to set a comment !")
If FILE$<>""
FILE$=FILE$+Chr$(0)
'**** Set comment ****
Dreg(1)=Varptr(FILE$)
Dreg(2)=Varptr(COMMENT$)
XCOMMENT=Doscall(-180)
'**** Error ? ****
If XCOMMENT=0:Print"Error occured!":Wait50:End If
End If
Edit

```

Listing 25

```

'*      Guru.AMOS      *
'*  ©1992 by Volker Steprath *
'*  Calls a DisplayAlert ( Guru Meditation ) *
Amos To Back
'**** Define text in display ****
T$=Chr$(0)+Chr$(75)+Chr$(38)+" No panic ... "
T$=T$+"This GURU was called up by AMOS !" + Chr$(0)
'**** Call up DisplayAlert ( Guru ) ****
Areg(0)=Varptr(T$)
Dreg(1)=80
XALERT=Intcall(-90)
If XALERT Then Print "left button" Else
Print "right button"
Amos To Front
End

```

Listing 26

```

'*      Info.AMOS      *
'*  ©1992 by Volker Steprath *

```



```

'* Shows informations about a DOS Disk *
Do
'**** Define drive ****
Input "Device.....: ";DEV$
If DEV$="" Then Edit
If Not Exist(DEV$) Then Edit
Dir$=DEV$
DEV$=Dir$
DEV$=DEV$+Chr$(0)
INFO$=Space$(40)+Chr$(0)
IN=Varptr(INFO$)
'**** Get lock ( ACCESS_READ ) ****
Dreg(1)=Varptr(DEV$)
Dreg(2)=-2
XLOCK=Doscall(-84)
'**** Get informatios ****
Dreg(1)=XLOCK
Dreg(2)=IN
XINFO=Doscall(-114)
'**** Unlock ****
XUNLOCK=Doscall(-90)
'**** Show informations ****
Print "Name.....: ";DEV$;Chr$(13)
Print "Errors.....: ";Leek(IN)
Print "Drive.....: ";Leek(IN+4)
N=Leek(IN+8)
If N=80 Then N$="Read only"
If N=81 Then N$="Validating"
If N=82 Then N$="Read/Write"
Print "Status.....: ";N;" ( "+N$+" )"
Print "Blocks.....: ";Leek(IN+12)
Print "Blocks used: ";Leek(IN+16)
Print "Bytes/Block: ";Leek(IN+20)
Print "Disktype....: ";
For I=24 To 27:Print Chr$(Peek(IN+I));:NextI
Print : Print "Disknode....: ";Leek(IN+28)

```

```

Print "Bytes free.: ";Dfree
Print "Full.....: ";Int(100-(100.0/856928.0)
*Dfree);"%
Print String$("-",39)
Loop

```

Listing 27

```

'*      Shade.AMOS      *
'*  ©1992 by Volker Stepprath  *
'* Change all colours of an IFF picture *
to black & White *
'**** Select IFF picture ****
FILE$=Fsel$("", "", "Please choose an IFF
picture !")
If FILE$<>""
'**** Load IFF picture ****
Load Iff FILE$,0
Wait Key
'**** Change colours ****
For I=0 To 31
C$=Hex$(Colour(I),3)-"$"
C=Val("$"+Left$(C$,1))+Val("$"+Mid$
(C$,2,1))+Val("$"+Right$(C$,1))
C$=Hex$(C/3)-"$"
C=Val("$"+(C$+C$+C$))
Colour I,C
Next I
Wait Key
End If
Edit

```

Listing 28

```

'*      Time.AMOS      *
'*  ©1992 by Volker Stepprath  *

```



```

'* Shows the actual time of system *
'**** Install buffer ****
N$=Space$(12)+Chr$(0)
'**** Get time of system ****
Dreg(1)=Varptr(N$)
XDATESTAMP=Doscall(-192)
'**** Calculate and show time ****
H=Leek(Varptr(N$)+4)
S=Leek(Varptr(N$)+8)/50
M=H mod 60
H=H/60
H$="00" : Right$(H$,Len(Str$(H))-1)=Str$(H)-" "
M$="00" : Right$(M$,Len(Str$(M))-1)=Str$(M)-" "
S$="00" : Right$(S$,Len(Str$(S))-1)=Str$(S)-" "
Print H$+" "+M$+" "+S$
End

```

Chapter 16

A beginner's guide to some AMOS commands and expressions used in this book.

It would make things a lot more complicated if every time a new command or name came up we stopped to explain it in detail, so we decided to add this chapter as a place where we can explain anything that we feel needs explaining. We shall assume that readers have very little or no knowledge of Basic terminology, so if you are a bit more advanced then you will be able to skip through this section.

What is BASIC?

BASIC stands for **B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode and as the name suggests, is the simplest form of coding and is therefore the most popular way of learning to program.

AMOS, Easy AMOS and AMOS Pro are all Basic languages that have many more advanced and flexible features than versions which came out before them.

Memory Banks.

These are the places where you can store the things you need in your program. These banks are reserved areas in the computer's memory that are then kept for a specific purpose. AMOS and Easy AMOS have 15 banks available, Pro has an unlimited number. You use these banks for storing your graphics music and data as well as for menus and work.

The files which contain these banks must have .abk at the end so that the program will recognise them as an AMOS bank. (abk means AMOS bank.)

Screens

An AMOS screen is like a page in a book, and is not just the TV or monitor screen you are looking at when you are playing games or programming. A screen can be larger than the area allowed on the monitor but to see all of it, you will have to make the program move the picture around. AMOS has eight screens available, numbered 0 to 7. The size of these screens and the number you can use at any one time will depend on how much memory your Amiga has available.

To make one of these pages available for use, it will be necessary to use the Screen Open command. This will reserve a space in the computer's memory for you to use.

To open a screen, you must tell the machine which screen you want to use, the size of that screen (width then height), the number of colours and the mode (i.e. the resolution) you wish to use.

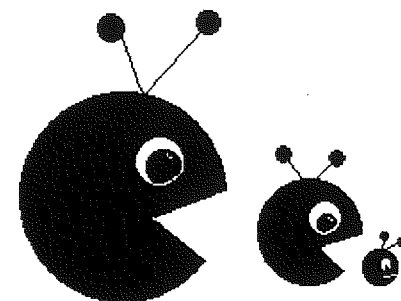
As an example, to open Screen 1 to a non-scrolling size suitable for NTSC users, in lowres mode that uses 16 colours, you would need the line:

```
Screen Open 1,320,200,16,Lowres
```

Ascii

This stands for American Standard Codes for Information Interchange. It is pronounced 'Asskey' and is simply a system which can be used to transfer data between different computers or to printers. It means that we can write the text for this book on either a PC or an Amiga and pass files from one to another as needed.

Bits and Bytes



1 Kilobyte = 100 Bytes = 8 Bits

A **bit** is the smallest piece of information that can be used in the computer's memory as either a 0 or 1.

A **byte** is made up of 8 bits. This is large enough to hold a single character, or a whole number that is equal to or less than 255.

A **Kilobyte** is made up of 1000 Bytes and is usually written as K - e.g. 10K

A **Megabyte**, or **Meg** to its friends is made up of 1000k. This is the measurement most commonly used when talking about the size of a computer's memory - An A1200 has a standard 2 megs of memory.

Bug

These are the annoying mistakes which are found when you test or run a program.

It is a slang phrase that came about in the very early days of computing when whole rooms were needed to house a computer. In those days moths and other insect beasties physically got into the wiring of the machines which caused them to crash. Hence the term 'bug' came into being.

Debugging is the slang expression used for the process of removing the

errors to allow a program to work properly.

Crash



What happens if a serious bug is found! In other words when the computer gets so upset it sulks, kicks out your program and gives you (usually) an error message.

Condition

Programs have to make decisions as to what action should be taken at different times. Put simply, it will have to answer true or false to a situation before deciding which action to take.

In practice this could be the user inputting a yes or no answer to a question and the program deciding what happens as a result of this.

Cursor

That is the thing that shows you where you are on the screen. This can be the mouse pointer, a dash or a block which flashes. The cursor keys are the arrows on your keyboard that allow you to move the cursor around the screen to whichever position you wish.

File

Computer files hold and store computer data in a single unit just as in an office, lots of letters can be stored in one cardboard file which means that the information is kept together to be recalled at a later time.

Font

This is the name for a style of text or type face which can be used in your programs. There are hundreds available from various sources and so you will be able to choose one which is the most suitable for the program you are writing.

This one is good for children

This one would make a good instruction screen.

THIS ONE WOULD GIVE ATMOSPHERE TO A SPOOKY GAME!

IFF

This means Interchangeable File Format and is the form in which DPaint for example saves out its files.

Label

This is a way of placing a marker in your program so that you can go back to a given point with a single instruction. You can use as many characters as you like in the label name, but it must always be followed by a colon (:) with no space between the name and the colon, e.g. labelname:

Pal

This is the standard used for TV pictures in the UK and Europe. If you open an AMOS screen to a depth of 256 pixels, it will fill the whole screen on a UK TV or monitor, but the same picture will have the lower 56 pixels missing if viewed in the USA or Japan where the standard is NTSC. If you make your screens 200 pixels deep, they will be fully viewable across the world!

Procedures

A procedure is a self contained mini program that is used to perform a specific task inside a main program.

The programs in this book are made up from procedures which means

that they can be cut out, adapted and used in other programs. Procedure writing is a neat way of programming as it is easy to find which bits of a program perform which tasks. You can give the Procedures names which remind you of what they do, e.g. Procedure SCROLLSCRN for the procedure that scrolls the screen, and so on. The procedures can be kept neatly together at the end of your listing and to make your listings even neater to scroll through, a procedure can be closed inside the editor so that only the name is visible, then opened up again if you need to work on it.

In the main listing, if you need the program to perform the task in one of the procedures, all you have to do is call it

e.g. SCROLLSCRN

Procedures mean that you can call up the routine at any time you need the program to perform the same task without repeating lines of code.

Parametered Procedures go a step further in that you can call the routine you need and give it different values, e.g. in the menu program, there is a procedure called `_FADEOUT[NBR]`. This will fade the colours of a screen out to black at a speed which you can set by calling the procedure as normal, but putting a number in place of the NBR in the square brackets. If you put in a 0 then the colours will turn instantly to black, a larger number gives a slower effect. NB The parameter MUST be in square brackets!

Scancode

Each key on the keyboard has its own numeric identity number, this is an alternative to the `Inkey$` command.

Directory

A directory is like the drawer in a filing cabinet where you can store things which belong together. Well-ordered disks - both floppy and hard, should be kept tidy so that you will be able to find the files you need with

ease.

A directory can be further divided into sub directories to keep things even tidier.

e.g. On your work disk you have all the files for your game, code, music, sfx, graphics, etc. all in the root (main) directory. It takes you ages to find a certain sound sample as you have to list the whole disk contents.

If you created a directory called Music, you could move all the music and sound effects into it so that you would know where to look. As an example of a Sub directory, once inside your Music directory you could create another called SFX and move all the Sound samples into it.

Classifying your graphics in a similar way will also tidy up the disk. Leave the main code in the root so that you will be able to load it quickly.

Syntax error

This is an error message that is displayed if you have made a mistake while typing in an Amos command, or if the computer does not understand what you have told it to do. It's like using bad grammar in an English exam - it gets marked as wrong! It is frustrating to see this come up, but at least it is a clue as to why the program is not working!

Strings

A string is a list of information which you can manipulate in your program. Strings are stored in String variables which have a '\$' after their name. This sort of variable can store both numeric and alpha characters. It is very important to learn how to use strings as it is almost impossible to write educational programs without this.

Variables

Simply put, a variable is something which can vary! Information needed in your programs are stored in variables which come in 2 varieties.

Numeric variables can only hold numbers, while string variables (they must have \$ at the end of their names) can have either numbers or letters in them.

```
A=2
B=3
C=A+B
Print C
```

This 'program' has all numeric variables and will add the value of A to that of B and then print the result on the screen as the value C.

By changing the values of A and B, you will get different results without doing anything to C. This is a variable used in its simplest form, but as you will see, you will be able to give something a different value or be able to perform an extra transaction on the result of a calculation without physically working it out for yourself, then putting it into your program.

```
A$="The cat "
B$="sat "
C$="on the mat."
Print A$+B$+C$
```

Here, the program will print The cat sat on the mat. If you change the order of the strings in the Print line, you will get a muddled sentence!

All programs have variables, you cannot get away from them! Amos allows you to choose variable names which are words, as long as the chosen word is not an AMOS command, therefore PRINT\$ would not be allowed as Amos would read it as the Print command, but PRNT\$ or _PRINT\$ would be accepted.

Development Time

This is a term which means the time during which you are creating your program rather than the time after which it released.

Runtime

This term is used to describe an event which takes place while the program is actually performing. For example, the crosswords in the jungle game in this book are created using a crossword generator which constructed the puzzles during the program itself. If the puzzles had been taken from a set of predefined puzzles, then they would have been created in development time.

A runtime disk is one which contains a program which will auto boot and does not need an operating system in order to work i.e. an Amos program disk which does not need Amos to run.

Loops

There are a few different types of loops on AMOS, but their function is broadly the same and that is to keep on repeating a task until a certain condition has been met. This could be to print something until a counter reaches 10, or to make the program wait until a mouse key is pressed. It is a way of getting the program to repeat a task with a simple instruction rather than you having to type ion line after line of code.

e.g.

```
For Z= 1 To 10
A$ = "Hello!"
Print A$
Next Z
A$="Hello!"
```

instead of

```
Print "Hello!"
Print "Hello!"
Print "Hello!" .....ten times in all!
```


Zone

Zones are rectangular areas which can be reserved on an Amos screen and set so that the program can detect when the mouse pointer or bobs etc. pass over it. Zones can be used as interactive buttons in programs where you need to get a response from the user, or in games to detect when a bob passes over an area of the screen. There is only one limit to the amount of zones you can set up on a screen and that is the amount of memory you have available.

There are several PD programs around that will grab zones and save out the data for you, using a zone grabber will speed up your programming, lots of programmers write their own so that they can get exactly the information they need.

Bobs

Bob stands for **Blitter Object** and is a moving graphic image which you can move around the screen without destroying the existing graphics. You can create bobs of any size and of any amount of colours, again, it the amount of available memory which decides how many you can display on a screen at any time.

Double Buffer

You will need to use a double buffered screen if you have moving images on the screen. A double buffered screen is actually two screens, one being an exact copy of the other. Think of it as one behind the other, the one in front is called the Physic (physical) and the one behind the Logic (logical). All the graphical changes take place on the logic screen, out of sight, then the whole screen is switched with the physic screen so that the user does not see any changes which would appear as flickering images on the screen. All the screen changing is done automatically when the Double Buffer command is used. A point to remember is that this system does take up twice the memory of a normal screen, but it is an essential part of creating Amos games.

Chapter 17

A Word About the Extensions available for AMOS.

At the beginning of this book we told you that we had not used any extensions in the programs for this book so that you could use the book and programs without needing anything more than your Amiga and Amos.

There are several extensions for Amos which we use all the time and would not be without as they make programming faster and easier. Many of you will own some of the extensions listed below, but for those who are just entering the world of Amos, here follows a brief description of what is available for Amos, we are not giving a full list of commands for each one as this would be a breach of copyright and we do not want to aid anyone who has 'borrowed' a friends copy and does not have the manual!

At the moment, all the extensions, bar one, mentioned here are for Amos Classic. Amos Pro, now has its own compiler, and the comments made regarding the Compiler for Amos Classic, also apply to that for Amos Pro.

What is an extension?

Amos has several extensions, these are programs which have been written either by Europress, who released Amos, or by other programmers who want to better Amos. A extension simply extends Amos's abilities by adding new sets of commands to those already included.

Some extensions are official, some are not. Official extensions are those which have been registered with Aaron Fothergill of Shadow Software, and given an official extension number.

An unofficial extension is one that, although legal in the eyes of the law,

has not been given an official extension number to attach it to Amos and the programmer uses whichever number he sees fit.

This might sound like a petty system, but it is sensible and should be adhered to. There is no cost for this service, so there is no excuse for programmers not to stick to the rules.

The conflict and confusion comes in when a programmer has an official extension attached to Amos, then finds an unofficial one that he really likes, and wants to use only to find that the unofficial one requires it to use the same extension number as the officially registered one. This has happened to us. We have always used Ctext in our programs, and we would not be without it, but we were sent a version of LDOS, a new, unofficial extension, to look at. LDOS is a good extension and has many features which we wanted to use, but it needed the same extension number as Ctext, so we had a decision to make. Ctext won and LDOS is not used. This is all because the programmer chose his own number instead of registering with Shadow Software and has probably lost the programmer many sales because of the conflict.

If you are a potential extension writer, then please bear this in mind and get an official number! If you need more information on this, please contact us or Aaron Fothergill.

AMOS 3D

This is an extension written for Amos Classic, but which, if you upgrade your Amos Pro to at least V1.12, will work with that as well. It allows the use of 3D objects on Amos screens and comes complete with its own object modeller.

AMOS COMPILER

This allows you to speed up your Amos programs and will also make them executable. Whether you are a serious Amos user, or whether you just use Amos for fun, this is a well worthwhile investment. There are

extra commands included in the compiler, and the most useful of these are SQUASH and UNSQUASH which will squash and unsquash banks better than Amos's PACK command.

CTEXT

Ctext is an extension by Aaron Fothergill of Shadow Software. It allows you to place multicoloured fonts up to the screen resolution being used, on to an IFF screen. It does this very quickly and has many commands to support its functions. With the newest Ctext (v2) you can also have multiple fonts in memory at the same time and it allows you to change the colour in a font in runtime.

DSAM

This is a brilliant sound extension for Amos. As you have probably found out, samples played in Amos have to be played from valuable Chip memory, but Dsam allows you to play it from Fast!

It will also play a sample which is stored on disk, not in the computer's memory. e.g. If you have a 600K sample on disk, you can play it without loading the whole file.

This extension contains many commands which will allow you to make the most of this valuable addition to the Amos family.

DUMP

This is an official Europress extension, the latest version does work with parameters which means that you can print just a section of the screen. This extension comes free of charge on the latest update to Amos from the AMOS PD Library.

LDOS

This is an unofficial extension that does quite a variety of things. There is a new Load and Save command which is faster than the Amos original,

and you can set protection and comment bits of a file. Also, you can decrunch Power Packer data files, manipulate Arrex and much, much more.

LSERIAL

This is done by the same author as the above and is also an unofficial extension. It is faster than the serial extension which comes with Amos.

RANGE

This is an official extension from Shadow Software and is a sort of Lucky Dip extension where Aaron Fothergill puts all the commands that he feels do not fit into any other of his extensions. Range is used very often in our programs, in fact there is probably something from this extension and Ctext, in everything we've written! Range's commands include SHUFFLE which is a random number generator something like the Procedure RDOM used in this book. RANGE is a command which forces a variable to stay between 2 values. CASE changes bob and icon colours, BHEIGHT and BWIDTH and many more.

STICKS

This is a relatively new extension at the time of writing (October 1993) which allows you to detect more than one button on the joystick and gives full joystick support. It's best facility, as far as we are concerned, is its ability to allow you to use two mice to control a game. That is, you can plug in two mice, and each of them will have its own mouse pointer with which to play the game. This extension was written by Nigel Critten, look out at the end of this chapter for details of his latest extension!

TOME

We know that we are known as TOME fanatics, and really that cannot be disputed! We think that TOME is the best Amos extension to date and this has nothing to do with favouritism towards the author, Aaron

Fothergill! We have used it in many of our projects, both in licenseware and in our commercial releases, and it has never let us down yet, even though we have taxed it to its limits at times! Some software houses use TOME to create their block map worlds for their games, so we are not alone in finding it very useful.

It can take a while to learn how to use TOME, but the time spent learning will pay off in the long term as when you can control it properly, it will save you more programming time than you spent getting to know it.

TOME is very good now, but Aaron has plans to make it even better when a new AGA extension comes out (see later). Aaron intends to use the AGA .lib to make TOME even faster!

TOME has to be seen to be believed and is not just for sideways Shoot'em-ups, but can be used to create dungeon master type worlds and lots more besides. The latest version, TOME Series 4, includes things like animated tiles which mean that you can have automatic spot anims, such as a constantly flickering torch, running with ease. It also has falling tiles such as you see in the old boulderdash games. Really, it's only your imagination that can hold you back!

ATOMIX

This is the one extension which at present is just for Pro users, although the programmers say that they intend doing an Amos Classic version as well. This is in fact the only official extension done by a third party for Amos Professional. Atomix is a very powerful data compression extension and is well worth looking at.

Latest news.....

Word has recently come in that there is to be a new extension for Amos which will be released by its author, Nigel Critten before Christmas 1993. This extension is for Amos Classic and will allow the full graphic manipulation for the AGA chipset.

Whether you have AGA or not, this extension is going to be a must as it will still display 16 or 32 colour graphics but will use commands which are much faster than their original counterparts.

There are going to be hundreds of commands in this extension, far too many to mention here but it is expected to include

- Opening and using intuition screens
- Applying IFF screens to intuition screens
- Grabbing intuition screens and putting them onto Amos screens
- It is hoped that the number of screens you can open in Amos will only be limited by the amount of available memory.
- Much faster bob and screen commands - every graphic command is to be altered to accept AGA which should speed up Amos whether AGA is used or not.

Chapter 18

Information about your free disk.

There is a coupon at the end of this book which you should use to send away for your free disk which contains the Amos in Education program, the graphics needed to write the House Counter game and a few other goodies which are described below.

The Amos in Education program is **source code only** so you will need to load it into Amos to see it run.

SpriteX V1.3 © Shadow Software

This, apart from the Amos in Education programs(!), is the star program on the disk. This program was previously distributed as Licenseware, but now, thanks to the generosity of Aaron Fothergill of Shadow Software, is being given away to readers of Amos In Education.



Please note that this program is **not Public Domain** so please do not give away copies to your friends. This version of SpriteX is the **only** version which is being distributed as a compiled program, so any copies found without the publisher's label will be seen as pirated copies!

The program has been compiled to Amos, partly for the reason above, but mainly to reduce the 'test' time when it is run.

That's the little lecture over, now let's tell you about SpriteX!

You might be asking yourself why we think you need another sprite editor when you already have the one given with Amos. The best way to answer that is to tell you to load the Amos sprite editor and use it for a while, then load SpriteX and try that out. You will probably never use the Amos one again!

SpriteX is an exceptionally good sprite and animation editor.

We shall give you a brief tour of SpriteX to show you what's what, but most of it is self explanatory and it will be easier to let you try it out for yourselves.

When you run the program, you will see a screen which is split into 4 distinct sections. The top section contains your control buttons. Down to the left is your magnified work area and next to this is the normal view area. To the right of these is your colour palette which shows your pen and background colour in long strips on either side.

The top row of buttons from left to right are:-



Load from disk,



Save to disk,



Append from disk (loads another bank of sprites from disk and adds it to the end of the current one.



SpriteX's bob cutter is next,

SWAP

Swap exchanges between bobs and icons.

At the end of the row are the 'Save Icon' button, 'Bob/Sprite' button which toggles between bobs and sprites, and the animator button. The animator allows you to animate your bobs so that you can check your work to see if any alterations need to be made.

The buttons below these should be familiar as they are the basic drawing tools found in many art packages.



The SpriteX Bob Cutter

When you click on this, the whole screen clears and a new panel appears which allows you to load an IFF picture and then cut the bobs out of the

picture in one of two ways. You can grab the bobs either using the same method used for grabbing brushes in art packages, or you can grab a whole line of bobs. The second method can save a lot of time if you have set up your IFF screen of bobs properly.

The best way of seeing this function at work is to create yourself an IFF picture in your favourite art package of, for example, 10 boxes side by side with a gap in between each and in different colours. Save the screen out and then load it back into the bob cutter area of SpriteX. Click on the 'Grab Line' button and drag out a box with the mouse pointer without releasing the button so that all your 'boxes' are covered. Release the button when they are covered and sit back and wait. SpriteX will now automatically cut out all 10 of your boxes and add them to your bob bank.

The best way to find out about SpriteX is to use it! It is user friendly and you will find that it is very easy to get to grips with.

Text Demo

This is a hybrid of the PRINTLINE procedure which is used throughout the programs in this book, but is much more involved and does a lot more.

This is a program which you can have great fun with if you play around with the parameters and see what happens! If you are intending to use these routines on a double buffered screen, make sure that Autoback is 0 when the procedure is called and then screen copy from the logic to the physic screen as can be seen in the PRINTLINE procedure.

Screen Effect

This is an example of a screen effect which gives the illusion of your picture being unrolled down the screen like a roller blind. It looks complicated, but it is very easy to create.

Hot Spot Placer Utility

This is our own development utility and is therefore not as user friendly as one written for general release! This has been used in every project we have done and saves a lot of memory.

It is used in conjunction with SpriteX, which is used to make the bobs as small as possible by pushing them to the top left of the of the box, and then to create the animation frames.

Most people enlarge the size of the bob and then move the image around in this area to get animations correctly placed. This wastes a lot of memory! The hot spot method works by placing the hotspot of the bob until all the frames are in line with each other ready for animation without taking up extra memory. These were then used to create the bob bank for the Amos In Education programs so that we could keep the memory usage as low as possible.

The controls for this program are included at the start of the listing as Rem lines. Remember that this is our personal utility and is not user friendly, but we hope that you will find it useful. This program can also be used as an X Y co-ordinate finder for bob placement.

Appendix

Table of Key State Values

Key	Key State	Key	Key State
Escape	69	[26
F1	80]	27
F2	81	a	32
F3	82	s	33
F4	83	d	34
F5	84	f	35
F6	85	g	36
F7	86	h	37
F8	87	j	38
F9	88	k	39
F10	89	l	40
`	00	:	41
1	01	#	42
2	02	z	49
3	03	x	50
4	04	c	51
5	05	v	52
6	06	n	54
7	07	m	55
8	08	,	56
9	09	.	57
0	10	BACKSPACE	65
-	11	DEL	70
=	12	HELP	95
\	13	RETURN	68
q	16	CUP	76
w	17	CDOWN	77
e	18	CLEFT	79
r	19	CRIGHT	78

Key	Key State	Key	Key State
t	20	ENTER	67
y	21	CONTROL	99
u	22	CAPSLOCK	98
i	23	LEFT SHIFT	96
o	24	RIGHT SHIFT	97
p	25	LEFT ALT	100
		RIGHT ALT	101
		LEFT AMIGA	102
		RIGHT AMIGA	103
Numberpad			
0	15	(90
1	29)	91
2	30	/	92
3	31	*	93
4	45	-	74
5	46	+	94
6	47	ENTER	67
7	61	.	60
8	62	SPACEBAR	64
9	63		

And finally.....

We would like to thank everyone who has helped and encouraged us to write this book.

Aaron Fothergill for giving us SpriteX to include on the free disk as well as his support.

The following programmers whose PD programs provided the procedures in Chapter 15, even if they are unaware of the fact that their work has been included!

Nick Simpson
Volker Steprath
David Tucker

And a couple of other programmers who were too modest to put their names in their listings!

Greetings and Best Wishes also go to

Ben Ashley, Steve Bennett, Paul Townsend, who are always there when needed.

Nigel Critten, for what looks to be a very exiting AGA extension.

All our Totally Amos members whose endless quest for Amos knowledge has inspired us to write books!

Melanie, Ben & Philippa who have to put up with the keyboard clicking while they are watching TV!

The Official

AMOS

VAT NO 557 9491 84

Public Domain Library

1 Penmynydd Road Penlan Swansea SA5 7EH

☎/Fax 0792 588156

Run by Anne Tucker

I want to write a Space Invasion game, but I can't get the enemies to move across the screen!

I am writing a demo, but I cannot see how to scroll a huge font!

We are trying to make the changes between screens more interesting, but how?

I need Sams!

I want some fonts!

I can't write the music!

I NEED SOMETHING TO GRAB SPRITES AND ZONE DATA FOR ME!

I NEED HELP!!!!

These are some of the many cries for help we get from AMOS users! We now have nearly 500 disks in the Amos Library, 90% of which have source code which you can load into AMOS to see how other AMOS users have tackled the problems you might be facing.

There are 40 disks of pure source code - collections of small programs collected from several programmers. There are now 4 disks of Totally AMOS programs written by TA readers as well as disks of source provided by individual programmers. As well as this we have loads of stand alone games and utilities for you to enjoy.

Disks are priced at just £2.00p each (plus 50p towards the postage per order) If you buy 10 disks, you may choose another 1 free of charge.

Catalogue Disks cost £1.00p (inc P&P)

Please send an SAE or IRC for an information sheet, order form and a printed lists of the latest disks.

Totally The Bi-monthly disk magazine **AMOS** For AMOS Users By AMOS Users

Join the band of Amos Users by subscribing to Totally AMOS!

We offer help on all AMOS subjects with help for programming problems, tutorials by other Amos Users which give a different view on various topics, music, sams and art. Readers are encouraged to send in their work so that other members can benefit from seeing the source code.

Totally AMOS is really aimed at beginners, with occasional articles for more advanced coders. We encourage readers to share their knowledge as well as their problems by telling us of the tricks and tips they have found as well as sending in programs for other people to see.

**A year's subscription costs £18.00p (UK)
£21.00p (Europe and ROW)
for 6 issues (P&P inc)**

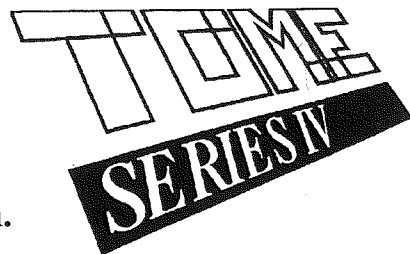
**Single Issues cost £3.00p (UK) and £3.25p (ROW)
(P&P inc.)**

**Please state from which issue you wish your subscription to start.
Issue 12 published in September 1993.**

Back issues always available!

**Issue 0, September 1991 is available as APD 341
Price £2.00p (+P&P)
Prices apply from 30-9-93**

"AMOS TOME has everything in it, I'll never use any other map system", Len Tucker, Programmer of Noddy's Playtime, author of AMOS in Education and AMOS in Action.



Well, Len Tucker thinks its the bee's knees !
What are you going to say about AMOS TOME ?

"Go and buy it now." ●●●●○ Phil South, Amiga Shopper magazine

"I can heartily recommend the AMOS TOME system to any serious AMOS user" Stephen Hill, AMOS Game Maker's Manual

"TOME.. has now established itself as the extension system no AMOS programmer can afford to be without." 88% Nick Veitch, CU Amiga Magazine

Written by Aaron Fothergill, Author of Jetstrike, and Editor of the Official AMOS Club, AMOS TOME is the ultimate extension for AMOS. TOME Series IV comes with 67 commands, TOME Editor (the most powerful available on any computer), 4 demo games (all sourcecode), 80 page manual and example programs for all the commands.

Also included with TOME Series IV is the latest version of the AMOS Club/Shuffle extension with a further 60+ commands!

AMOS TOME works with AMOS 1.36 and the Compiler and with all Amigas including the 1200 and 4000. TOME Does not support AMOS Professional.

Full user support for AMOS TOME is constantly available through the Official AMOS Club, and through Totally AMOS disk magazine, so it is the most widely supported AMOS Add-on

Please Send me
AMOS TOME Series IV ☐ £29.99

Name.....

Address.....

.....Postcode.....

Make cheques (in pounds sterling only please) payable to "The AMOS PD Library" and send orders to **Totally AMOS, Dept AIE, 1 Penmynydd Road, Penlan, Swansea. SA5 7EH**

AMOS TOME is (c) Shadow Software 1989-93



Get your free copy of
the
AMOS IN EDUCATION
Program

Complete source code as well as other
useful utilities and SpriteX © Shadow
Software.

Return this form to:

Kuma Computers Ltd
12 Horseshoe Park
Pangbourne
Berks
RG8 7JW

Name:

Address:

Overseas requests please enclose £1.20p to cover P&P

5

AMOS In Education

AMOS in Education has been written to help AMOS programmers to understand what is needed to produce an educational program. It gives all the information needed to produce an educational program from the initial planning stages to tips & tricks needed to make your program run smoothly. It starts by giving three full specifications, graphics and code for three games and the menu, then allows you to practice writing for yourself, just the graphics and specs are given for a further section. For those who want to go a step beyond this just the game specs are given for a last section for you to add your own graphics and ideas before linking it to the main program.

As computers feature so highly in the National Curriculum in schools, it is very useful to be able to write programs which children can use at home which will reinforce what is being learned at school. As there are more children than computers in most classrooms, many children do not get enough time to practice using them.

As parents ourselves, we think that home computers should be used to educate as well as entertain children and so try to make learning fun in the programs we design and write. Children will learn faster if they are enjoying *playing* as well as working! In our work with children at a local school we were surprised to find that the children volunteered the information that they wanted to do more with their computers than endlessly fight aliens, but many of them have no other choice as most educational programs are out of the range of their pocket money.

AMOS gives people the opportunity to write educational programs for their children, brothers, sisters or friends. Even if no-one else ever uses the program, it will always be special for user for whom it was written!

The information in this book has been gathered by Anne and Len Tucker during the last three years of programming with AMOS. During that time they have produced several licenceware educational titles before working with commercial software houses in the production of educational packages.

A coupon is included in this book so that you can send for a FREE disk containing the finished programs, useful routines and all the source code for you to look at and follow as you make your way through the book.

Published by
Kuma

Kuma Books Ltd,
Pangbourne, Berkshire, England
Tel: 0734 844335
Fax: 0734 844339

£12.95

ISBN 0-7457-0225-2



01295 >



9 780745 702254